Systematic Support for Accountability in the Cloud

Winai Wongthai

Newcastle University

Newcastle upon Tyne, UK

In Partial Fulfilment of the Requirements for the Degree of

Doctor of Philosophy

October 2014

To Mum, Dad, and my Family

Acknowledgements

I would like to express my gratitude to Professor Aad van Moorsel for his supervision throughout my PhD. His guidance and support are the foundations of this thesis. I would also like to thank those people I have been fortunate to work with throughout my PhD: Dr Feng Hao, Dr Nick Cook, and Emeritus Professor Santosh Shrivastava. Additionally, I would like to express my appreciation for the love and support of my family, for their help and support throughout my PhD. Finally, I would like to thank all people in the cyber crime lab, particularly Francisco Liberal Rocha, and in the computing school, for their help and support.

Abstract

Cloud computing offers computational resources such as processing, networking, and storage to customers. Infrastructure as a Service (IaaS) consists of a cloud-based infrastructure to offer consumers raw computation resources such as storage and networking. These resources are billed using a pay-per-use cost model. However, IaaS is far from being a secure cloud infrastructure as the seven main security threats defined by the Cloud Security Alliance (CSA) indicate. Use of logging systems can provide evidence to support accountability for an IaaS cloud.

An accountability helps when mitigating known threats. However, previous accountability with logging systems solutions are provided without systematic approaches. These solutions are usually either for the cloud customer side or for the cloud provider side, not for both of them. Moreover, the solutions also lack descriptions of logging systems in the context of a design pattern of the systems' components. This design pattern facilitates analysis of logging systems in terms of their quality.

Additionally, there is a number of benefits of this pattern. They could be: to promote the reusability of design and development of logging systems; that designers can access this pattern more easily; to assist a designer adopts design approaches which make a logging system reusable and not to choose approaches which do not concern reusability concepts; and to enhance the documentation and maintenance of existing logging systems.

Thus, the aim of this thesis is to provide support for accountability in the cloud with systematic approaches to assist in mitigating the risks associated with real world CSA threats, to benefit both customers and providers. We research the extent to which such logging systems help us to mitigate risks associated with the threats identified by the CSA. The thesis also presents a way of identifying the reference components of logging systems and how they may be arranged to satisfy logging requirements. 'Generic logging components' for logging systems are proposed.

These components encompass all possible instantiations of logging solutions for IaaS cloud. The generic logging components can be used to map existing logging systems for the purposes of analysis of the systems' security. Based on the generic components, the thesis identifies design patterns in the context of logging in IaaS cloud. We believe that these identified patterns facilitate analysis of logging systems in terms of their quality.

We also argue that: these identified patterns could increase reusability of the design and development of logging systems; designers should access these patterns more easily; the patterns could assist a designer adopts design approaches which make a logging system reusable and not to choose approaches which do not concern reusability concepts; and they can enhance the documentation and maintenance of existing logging systems.

We identify a logging solution which is based on the generic logging components to mitigate the risks associated with CSA threat number one. An example of the threat is malicious activities, for example spamming, which are performed in consumers' virtual machines or VMs. We argue that the generic logging components we suggest could be used to perform a systematic analysis of logging systems in terms of security before deploying them in production systems.

To assist in mitigating the risks associated with this threat to benefit both customers and providers, we investigate how CSA threat number one can affect the security of both consumers and providers. Then we propose logging solutions based on the generic logging components

and the identified patterns. We systematically design and implement a prototype system of the proposed logging solutions in an IaaS to record history of customer's files.

This prototype system can be also modified in order to record VMs' process behaviour log files. This system can record the log files while having a smaller trusted computing base, compared to previous work. Additionally, the system can be seen as possible solutions that could tackle the difficult problem of logging file and process activities in the IaaS. Thus, the proposed logging solutions can assist in mitigating the risks associated with the CSA threats to benefit both consumers and providers. This could promote systematic support for accountability in the cloud.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The National Institute for Standards and Technology or NIST [4] defines cloud computing as:

"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Many people argue that the cloud is the future of computing, and has the potential to transform the IT industry in a wide variety of application areas [5]. This thesis focuses on public cloud computing, where the cloud infrastructure is provided for open use by the general public, and owned, managed, and operated by business organisations which are called cloud providers [4].

Figure 1.1 illustrates a public cloud computing environment. In the figure, there are two main sides or parties involved in the environment: the customers and the providers. The provider owns its cloud infrastructure which includes a virtual machine manager and cloud offering services or products such as servers and networking.

Some of the services may require the provider to store and operate a customer's

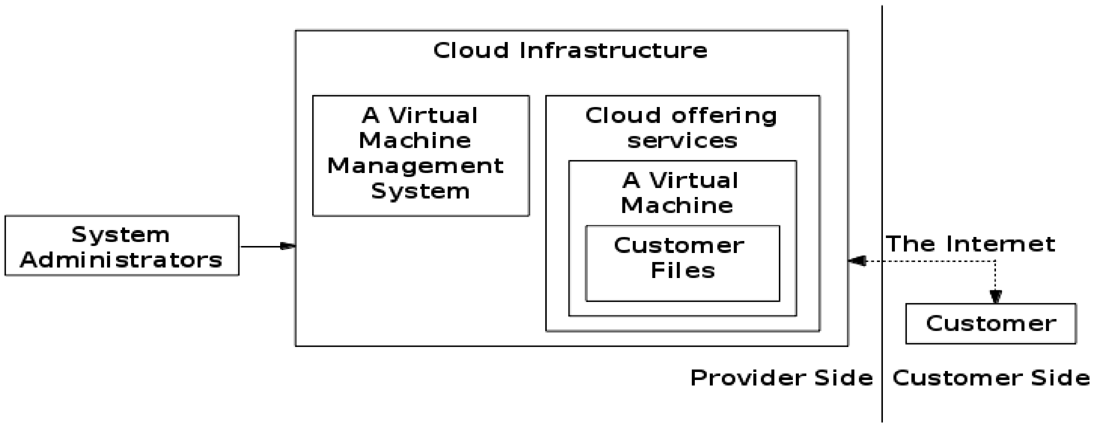


Figure 1.1: A public cloud computing environment

files, for example executable programs and/or database files, in a virtual machine or VM[1]. A virtual machine is managed by a virtual machine management system which is controlled by a provider. Full details of the management system are in Section 2.2. The cloud infrastructure is maintained by the provider's employees such as system administrators. Customers can purchase or rent these services, and use or interact with their services via the Internet. This thesis uses the term 'cloud' to refer to 'public cloud computing'.

The benefits that the cloud can bring to the business, education, government, and science industries are example [6; 7; 8; 9; 10; 11; 12; 13; 14]. For example, to be convenient and to minimise the budget for highly data-intensive computing (such as in the fields of physics, climate science, or astronomy), scientists can use cloud to share their data [15]. In education, for example, the european project Cloud Services for E-learning in Mechatronics technology or CLEM [16] is researching how advanced mechatronics facilities (such as highly sophisticated robotic laboratories which are not available to all colleges and schools) can be delivered through the cloud.

The cloud can provide for the needs of the IT department with low investment and unlimited resources [5]. In addition, it can provide the ability of easily adding or removing resources on demand, scalability, availability and/or reliability for

[1]It is a software that can replicate a physical computer system which includes an operating system or OS, a main memory, disks, networking, etc.

the IT environment with cheap cost. These attributes make the cloud attractive to IT departments.

IaaS cloud. Note that, the terms consumer and customer are used interchangeably throughout this thesis. There are many types of public cloud. We focus on infrastructure as a service or IaaS public cloud such as Amazon Elastic Compute Cloud or Amazon EC2 [17]. IaaS consists of a cloud-based infrastructure to offer consumers raw computation resources, such as storage and networking, with a pay-per-use billing model. This type of cloud is used by enterprise, government, and academia, such as in medical experimentation [18; 19; 20; 21]. Figure 1.1 can also represent an IaaS cloud environment. From the figure, customers can rent VMs, and then they can virtually own and control these VMs. Following this, they may also upload and store their files in these VMs.

The problems of the cloud. Trust of consumers in a cloud/IaaS is extremely important for its continued proliferation. Customers want to know how and where their data and related logs are stored and who has access to them. There are many causes of security concerns. For example, [9] argues that cloud security concerns (e.g., virtual machine/VM-level attacks) involve computer and network intrusions or attacks that will be made possible or at least easier by moving to the cloud.

The Cloud Security Alliance (CSA) has extensively published on security concerns, see for instance the Security Guidance for Critical Areas of Focus in Cloud Computing report [22] and the Top Threats to Cloud Computing report [23]. The latter document is used in this study, providing the basic threats for which we aim to provide systematic support for accountability to mitigate the risks associated with these threats.

Cloud security concerns have to be solved before effective realisation of the benefits [24]. Thus, we use the CSA top threats report as real world concrete cloud problems that we intend to deal with. CSA continues working to focus on security concerns for the cloud after this report by publishing other reports such as Top Threats to Cloud Computing Survey Results Update 2012 [25], or The Notorious Nine: Cloud Computing Top Threats in 2013 [26].

Accountability with logging systems as the solution to mitigate risks associated with CSA threats. Although by itself it is not considered sufficient to avoid or remove the risks associated with threats, a logging system is an important aspect of any accountability solution in the cloud, as argued by many researchers [5; 8; 9; 10; 27; 28; 29]

Accountability in the cloud. To mitigate the risks associated with CSA threats, accountability is needed in the cloud. This means that the cloud behaviours can be inspected by any party or that the cloud is accountable, as argued by many researchers [5; 8; 10; 28]. To enable accountability in the cloud, [8] proposes concepts called an accountable cloud, outlines the technical requirements for an accountability, and states that the accountable cloud needs to have accountability. [9] gives almost the same idea as the accountable cloud.

Then [8] describes that an accountable cloud is the cloud in which (i) the consumers can investigate whether the provider is processing their data according to the agreement, (ii) if errors are reported, the provider can provide the evidence to verify who is responsible for the problem, and (iii) if there is an argument, the provider can present proof to the third party. Therefore, an accountable cloud could be a very good concept to deal with the CSA threats in IaaS.

Logging systems in IaaS. In this thesis, a logging system in IaaS is a system that resides in the provider's cloud infrastructure, and can collect and store log information of the infrastructure's components. One example of these components can be a virtual machine. Thus, a logging system is composed of logging processes and log files. A logging process performs logging tasks, whereas log files are used for storing contents produced by the logging processes.

Each logging process has its own task(s). For example, a main task of a logging process called Flogger [30] is intercepting the file operations of a virtual machine or VM for the purposes of accountability in the cloud. It manages to store such intercepted data as log files. We argue that logging systems are a core component for cloud monitoring/accountability, which can assist in mitigating the risks associated with CSA threats, as also argued by [23], [27].

Moreover, cloud-related research or projects consider logging systems as one of the main elements in their projects. For example, [31] argues that to enable true traceability of data in the cloud, it needs to have a logging mechanism which tracks and records data life-cycles and movements in the cloud. [32] (Distributed Management Task Force, Inc.) considers logging as one of the high-level requirements that a cloud service provider should make available as a part of their cloud service offering.

The Technology Strategy Board trust domains project [33] is working on providing a framework for modelling and designing e-service infrastructures for the controlled sharing of information. These infrastructures also include the cloud. This project provides five trust building primitives for the trust domains. Two of them are: accountability which is whether a person is accountable for actions at an organisational or individual level; and auditing which is the requirement to provide evidence of processing to third parties. Both primitives require logging to collect evidence or log files.

Thus, logging systems can be key elements of accountability in the cloud. To achieve trust and security in the cloud, accountability in the cloud requires a holistic approach [29], encompassing legal mechanisms, regulatory mechanisms, and technical mechanisms. One of the classes of technical mechanisms for accountability is detective controls. These controls can be used to analyse the incident of a privacy or security risk that goes against the privacy or security policies and procedures [27]. Transaction logs is one of the aspects of detective control [29]. Thus, the logging systems, ability to produce the transaction logs can be seen as an important aspect to be considered to enable accountability.

Systematic support for accountability in the cloud. The meaning of **systematic** from a dictionary is: using a fixed and organised plan [34]; done or acting according to a fixed plan or system [35]; and done according to a careful plan and in a thorough way [36]. The common key word from the dictionaries is a *(fixed or careful) plan.*

We argue that systematic support for accountability in IaaS includes three aspects. The first is simultaneous consideration of both the customer side and the provider side, which can have effects caused by CSA threats; thus, the logging solutions should be able to deal with the threats in order to benefit both sides simultaneously. The second is reducing the trusted computing base or TCB size of a logging system when building and deploying it in IaaS. TCB is a term in computer security to refer to the set of all hardware, software, and procedural components, which enforce the security policy [37], more details can be found in Section 2.13. To break security, an attacker has to subvert one or more of those components [37].

[38] argues that the TCB size of a software system (a logging system is also a software system) can be used to evaluate the trustworthiness of that system. [39] state that for any software system, the size of the system's TCB should be as small as possible. Thus, the TCB size of a logging system can be used to evaluate the trustworthiness of this system as well, more details are given in Section 2.13. The third is that the security of the logging systems themselves needs to be analysed before the deployment. All the systematic aspects will be summarised in the research gaps section below (Section 1.2).

We use the word **systematic** to refer to the planned introducing of logging. For example, in order to build logging systems in IaaS, the first systematic aspect is that the security of the logging systems themselves needs to be analysed before deploying the systems in the IaaS real world productions. The security concern can be the integrity of the log files that is produced by these logging systems, more details are given in Section 2.11. This is because the log files may be used in law courts; thus, the integrity of these files needs to be ensured before using them.

1.2 Research Gaps

This section summarises the research gaps of previous work or approaches.

- **Gap 1: Lack of systematic approaches to build logging systems in IaaS.**
  - **Gap 1 (a): Lack of simultaneous consideration of both customer and provider side**

    Research that proposes logging solutions for accountability such as [5; 8; 9; 27] usually discusses security in the cloud from a customer perspectives, which involves for example integrity and safety of customers' data. They do not explicitly discuss how the logging solutions can assist in security concerns from a provider's perspective such as from Figure 1.1 when criminals use a virtual machine to attack a virtual machine management system.

CSA continues to provide a new survey report (CSA Top Threats Cloud Computing Survey 2012 [25]) to investigate if the seven top threats [23] are still relevant. In addition, the report also provides an update on newly emerged threats. This survey report includes both cloud consumers and providers as the respondents: 53% of the cloud consumers; and 30% of the cloud providers. Although the report does not discuss solutions to deal with the threats, it is a good example that the research should consider the threats to cloud from both customer and provider perspectives. Accordingly, research should also provide the solutions for both sides.

Thus, it will be beneficial to discuss the effects to both sides, and to provide solutions to mitigate the risks associated with the CSA threats concerned to both sides.

– **Gap 1 (b): Lack of concerns of reducing the trusted computing base or TCB size of a logging system**

  A logging system is a software system which will not be run in an individual private organisation's perimeter, but in the provider's virtualised infrastructure or the IaaS cloud. This makes evaluation of the trustworthiness of this system more difficult than a software system under a private organisation. The logging systems need to be trustworthy.

  Thus, the size of the TCB can be a very important aspect, in order to propose logging systems in IaaS. For example, the proposed systems in [38; 39; 40; 41] are concerned with reducing the TCB sizes along with their proposed systems. Moreover, recent work that extensively relates to a secure cloud computing environment, such as that of [42] has also focused on reducing TCB size in their proposed architecture. Thus, we argue that consideration of TCB is important, when proposing logging solutions to deal with the IaaS issues.

– **Gap 1 (c): Lack of security analysis of the logging systems themselves before the deployment of the systems in to the IaaS real world productions**

  We believe it is important to perform a security analysis of the logging systems themselves before the systems' deployment in to the IaaS real world production. Present works [8; 9] give an outline of accountability/monitoring-related solutions for the cloud. These works involve the logging process and the log file components, which will be deployed in the cloud provider's infrastructure. However, they do not discuss the exact locations in the infrastructure that these components will be deployed such as: whether these are in a virtual machine management system which is controlled by a provider or a virtual machine which is controlled by a customer; and who will manage these components in these particular deployment locations and why.

  Thus, the deployment locations of these components directly and significantly affect the components' security concerns, for example how to ensure the integrity of the log file components when they are deployed in a virtual machine management system which is controlled by providers who may alter the log files to benefit themselves. This issue is fully discussed in Chapter 3.

  Other works that involve design and implementation of the logging solutions which could be applied in the cloud include [27; 39; 43; 44]. These works focus on monitoring and logging tasks. Ideally, a logging system can be analysed with respect to its features and achievable goals before deployment. However, the authors do not give concerns about the security analysis of the logging systems themselves, which include the logging processes and the log files before deploying the systems to the real world IaaS. Thus, it is important to perform a security analysis of the logging systems themselves, as will be fully discussed in sections 3.4.2.4, and 5.10.

- **Gap 2: Research that relates to logging in IaaS only focuses on system-centric logs**

  To build logging systems in IaaS, it is also important to consider log file types which can be produced by logging systems. There are two main types of log files as discussed in [30; 45]: *file-centric logs* and *system-centric logs*. The first traces files from the time they are created to the time they are deleted as discussed in [27; 45]. The latter focuses on the hardware layer log such as memory use, disk storage, temperature, voltage, etc, as argued by [46]. This also can concern event logs, user account activity logs, processor usage, etc, as argued by [30].

  [27] argues that research focuses instead on system-centric logs (e.g., memory use or disk storage) rather than file-centric logs (tracing customers' files in VMs from the time they are created to the time they are deleted). For example, [46; 47] provide a good set of recommended monitoring principles that should be performed by a provider to collect mainly system-centric logs. However, will the provider really implement these recommendations and share the monitoring reports with the customers? [48] argues that system-centric logs are usually disclosed to consumers.

- **Gap 3: Lack of analysis of what the contents of the log file should actually be, and of how the contents can be used to deal with the real world CSA threats to benefit both sides in detail**

  For example, previous logging works such as [5; 8; 9; 27] do not clearly discuss in detail what the log files' contents should be, and how contents can mitigate the risks associated with threats to benefit both customers and providers. What the content should be is dependent on what the problems (e.g., the CSA threats) are. The right log file content can solve the right problems.

- **Gap 4: Lack of descriptions of logging systems in the context of design patterns of the systems' components**

  A design pattern is a documented best practice or core of a solution that has been applied successfully in multiple environments to solve a problem that recurs in a specific set of situations [49]. Based on designing object-oriented software as design patterns, Gamma et al [50] discuss a number of benefits of the design patterns. For example, they state that: design patterns make it easier to reuse successful designs and architectures, expressing proven techniques as design patterns makes them more accessible to developers of new systems, and design patterns assist a designer gains a design right faster.

The design patterns in logging context could bring the same benefits as above. This could lead us to a spectrum of patterns for describing how to construct logging systems with varying characteristics. Previous work neglects to describe the logging component (e.g., logging processes and log files) configurations available as patterns and focus only on the implementation and evaluation of a single logging system. As a result, proposing, implementing, and evaluating logging patterns systematically have not yet been described in the literature.

This thesis concerns the broader contribution expanding from the components of logging to the subsequent patterns these components may create (each pattern with its own advantages and disadvantages).

1.3 Aims and Objectives

To address the research gaps in Section 1.2, this thesis aims **to provide support for accountability in the cloud with systematic approaches to mitigate risks associated with real world CSA threats, to benefit both customers and providers**. To achieve this goal, the project aims to satisfy the following objectives.

Objective 1: Understand the real world cloud problems (the CSA threats), and accountability with logging approaches in the IaaS

Objective 2: Design a generic framework of logging solutions to mitigate risks associated with the CSA threats

Objective 3: Define, identify, and draw conclusions on the advantages and disadvantages of logging system patterns, then analyse existing works in relation to the patterns

Objective 4: Design and implement logging systems based on the generic framework from Objective 2 and the identified patterns from Objective 3 to mitigate risks associated with a specific threat (threat 1 which is *abuse and nefarious use of cloud computing*, for example when cloud customers rent VMs then use these VMs to conduct criminal activities), and to the benefit of both cloud customers and providers

Objective 5: Evaluate the proposed generic framework from Objective 2, and the proposed logging systems from Objective 4

We evaluate the framework by demonstrating instantiations of logging solutions to deal with CSA threat 1 in Chapter 3 (the spamming case study) and in Chapter 6 (a proposed logging system to benefit both customers and providers). We then evaluated the performance of the proposed logging system in terms of the accuracy of the system in Chapter 6.

1.4 Methodologies

To satisfy Objective 1 and 2:

1. We use CSA threats as concrete cloud problems, and use Xen's [51] structure to replicate IaaS infrastructure. Then we investigate the components of IaaS structure and of existing logging systems in IaaS from previous work.

2. We investigate the importance of security of the logging systems which we consider as one aspect of systematic approaches in order to develop the systems in real world IaaS.

3. We investigate the logging process and log file components of each work. We locate these components into the replicated IaaS architecture. Then, generic logging components are proposed.

4. In order to demonstrate how the proposed generic logging components can facilitate analysis of logging systems in terms of security before deploying them in production systems, we map existing logging systems and our own logging system to the proposed generic logging components.

To satisfy Objective 3:

1. Based on the designed generic framework from Objective 2, we investigate all possible architectures of a logging system in an IaaS.

2. Associated with these possible architectures, we then define, identify, and give conclusions on the advantages and disadvantages of logging system patterns. To do so, first of all, this thesis discusses a definition of a pattern in general and in object-oriented software design and defines a pattern in logging system design and development. It then identifies and discusses our own three identified patterns for logging systems. After that, it gives conclusions on the advantages and disadvantages of the patterns.

3. This thesis analyses existing logging systems in relation to the identified patterns based on the given conclusions on the advantages and disadvantages of the patterns above.

To satisfy Objective 4 and 5:

1. To analyse how threat 1 (criminal activities in customer VMs) affects both sides, we investigate how customer VMs can attack other customer VMs, and also attack the provider's virtual machine management system (called dom0) in detail. We also carefully analyse vulnerabilities of the provider's dom0, which may lead to compromising the dom0 itself, and subsequently, all the customer VMs hosted by this dom0.

2. We discuss what contents of the log files can deal with the effects above. To demonstrate logging solutions regarding the file-centric logs, and to benefit both sides, we indicate the importance of customers' critical files in VMs. Then, we discuss the concepts of the history of critical files, the process behaviour log files, and the content of both types of log files. We then discuss how these log files can assist in mitigating risks associated with threat 1 stated above to benefit both sides.

3. This thesis then describes the proposed logging system architecture to obtain these log files based on: the generic logging components, its systematic

design and implementation, and the identified patterns. Based on the related components of logging systems from the generic logging components, we discuss how the proposed system can obtain the log files we need, while yielding smaller TCB compared to previous work.

1.5 Thesis Contributions

This thesis provides a number of key contributions to address the research gaps, and to satisfy the aims and the objectives stated above:

1. **An in-depth background and literature review of CSA threats and systematic support for accountability**

   This contribution addresses research Gap 1. The background and literature review summarises the key concepts: IaaS, CSA threats to IaaS, accountability/logging approaches to mitigate risks and the problems of the approaches, and systematic support for accountability in the cloud.

   This summary differs from previous work which focus on dealing with cloud problems without full consideration of the main involved/systematic aspects to provide logging systems. For example, previous work does not fully discuss the security analysis of logging systems themselves before deploying them into the real world IaaS productions. Without the consideration of the systematic aspects, it is difficult to efficiently and effectively enable logging systems.

   The value of the systematic approach is to provide clear visions of logging systems, development in the cloud, such as the security analysis of the systems. Then, the systematic approach can efficiently and effectively enable accountability in the cloud. This is because accountability in the cloud is the most important concept to assist in mitigating the risks associated with CSA threats.

2. **Generic logging components for IaaS cloud**

   This contribution addresses research Gap 1 (c). To facilitate systematic support for accountability in the cloud, generic logging components provide ways to build logging systems. The value of theses components is to encompass all possible instantiations of logging solutions for IaaS cloud, and to provide a clear view of all components that relate to logging systems in IaaS. This view provides a basis for the analysis of logging systems, security before deployment.

Thus, the generic logging components enable logging systems to be appropriately designed or manipulated by participating cloud parties such as a provider, customer, or auditor. As result, this enhances systematic support for accountability in the could.

3. **Analysis of how CSA threat 1 affects both the customer and provider side simultaneously; and the proposed logging solutions to assist in mitigating risks associated with the threat for both sides**

   This contribution addresses research Gap 1 (a). The analysis illustrates how CSA threat 1 (mis-usage of customer VMs) can affect both sides. This analysis differs from previous work which usually focuses on the effects of the threats to only either the customer side or provider side, providing solutions for either side, but not for both.

   The value of the combined analysis is to provide a basis to understand what the contents are that logging solutions need to collect to be used as evidence to deal with threat 1 for both customers and providers.

4. **The design of our proposed logging system yields a smaller trusted computing base or TCB size compared to previous work**

   This contribution is that the size of the TCB for the design of our proposed logging system is smaller than TCB sizes of the proposed logging systems of previous works. This contribution addresses research Gap 1 (b).

   All logging related components, for example an introspection tool and logger application, deployed in building the proposed logging system are inside dom0. Thus, the proposed system in our prototype implementation yields a smaller TCB while obtaining the history of critical files. This is because the TCB of our architecture includes only a hypervisor[1] and dom0, not a customer VM. In contrast, previous work have placed some of their logging related components in customer VMs. Other previous works that deploy the same introspection tool as we used yield the same TCB as ours. However, they are not designed to obtain the history of critical files.

5. **Collecting file-centric logs rather than system-centric logs**

   This contribution is that our proposed logging solutions focus on collecting file-centric logs rather than system-centric logs. This is because research Gap 2 states that current logging research in the cloud focuses on system-centric logs, while neglecting file-centric logs.

   There are some logging solutions that emphasise file-centric logs with an interception approach. However, the prototype implementation of the proposed logging system can be an alternative approach to collecting file-centric logs to enhance accountability in IaaS. This approach facilitates introspection of customer VM's memory from dom0. The introspection traverses the kernel data structures in the memory.

   The prototype implementation of the proposed logging solutions can collect a file-centric log history of customers' critical files. The history information is of file-centric logs rather than system-centric logs. The file-centric logs can present the associations between a process and files in a customer's VM, for example, a record of a process P reads file F. The proposed log files differ from previous works which focus only on system-centric logs including the connection topology, bus speeds, and processor loads.

6. **Presentation of how the results from the proposed logging system assist in mitigating the risks associated with threat 1**

   This contribution is that we provide many scenarios to present how the results from the prototype implementation can be used to form log files to assist in mitigating the risks associated with threat 1 for the benefit of both customers and providers. This contribution addresses research Gap 3.

[1]software running on a physical machine and allows the machine to run multiple OSs at the same time [52]

To assist in mitigating the risks associated with threat 1 to benefit customers, we discuss formation of a history of critical files from the results of the prototype implementation. This includes analysing nine scenarios of malicious incidents in a customer VM when, for example, a customer shares her VM with other users.

To assist in mitigating risks associated with threat 1 (e.g., when criminals use VMs to conduct spamming activities) to benefit providers, we discuss the formation of process behaviour log files. This shows that our proposed solutions can assist in mitigating the risks associated with real world IaaS issues and many scenarios, and can benefit both customers and providers.

7. **Three proposed design patterns in the context of logging in IaaS cloud**

   This contribution addresses research Gap 4. The proposed patterns facilitate analysis of logging systems in terms of their quality. These patterns could increase reusability of the design and development of logging systems. Designers should access these patterns more easily. The patterns could assist a designer adopts design approaches which make a logging system reusable and not to choose approaches which do not concern reusability concepts. The patterns can also enhance the documentation and maintenance of existing logging systems.

   We provide a spectrum of patterns for describing how to construct logging systems with varying characteristics. For developers, when building a logging system, the knowledge of characteristics of this system could assist them to get the right design of the system with minimal effort and time commitments. We also clarify why a number of patterns and logging system architectures based on these patterns are missing. To the best of our knowledge, these three logging patterns are not yet described in the literature.

1.6 List of publications

- W. Wongthai, F. L. Rocha, and A. van Moorsel. A generic logging architecture template to mitigate the risks associated with spam activities in infrastructure as a service cloud. In a Poster Session at The Newcastle Connection 2012, August, Newcastle, UK. [53]

  This work is mainly from Chapter 2 and 3. It is the initial work which shows that, in an IaaS environment and based on logging components, it is possible to obtain log files that can be used as evidence to assist in mitigating the risks associated with spamming activities performed by a malicious customer using his VM.

- W. Wongthai, F. Rocha, and A. van Moorsel, "A generic logging template for infrastructure as a service cloud". In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, 2013. [54]

  This work is mainly taken from Chapter 2 and 3. It extends the logging components above to be a more generic version as generic logging components. It discusses how the generic logging components facilitate the analysis of logging system security in IaaS cloud.

- W. Wongthai, F. Rocha, and A. van Moorsel, "Logging solutions to mitigate risks associated with threats in infrastructure as a service cloud". In tracks of service and application in 2013 International Conference on Cloud Computing and Big Data (CloudCom-Asia), Fuzhou China, December 16-19, 2013. [55]

  This paper presents concepts of logging solutions (Chapter 5 and 6), such as the history of a customer's critical files and how these can assist in mitigating the risks associated with threat 1 to benefit both customers and providers.

1.7 Thesis structure

The remainder of this thesis is structured as follows. Chapter 2 presents the background and literature review to provide the context for this thesis. Chapter 3 presents the generic logging components for IaaS, and case studies of how these components work. Chapter 4 then describes the related work in the form of logging patterns. Chapter 5 presents how threat 1 harms both customers and providers, and proposes systematic support for accountability as a solution to mitigate risks associated with the threat. Chapter 6 is the implementation of the proposed solutions, and the discussion of how we obtain the results, and how the results can assist in mitigating the risks associated with CSA threats, and additionally a comparison of our proposed system with previous work. Finally, the conclusions and future work are presented in Chapter 7.

Chapter 2

Background And Literature Review

The main contribution of this chapter is to present a background and literature review for the thesis. This chapter discusses the definitions of cloud computing and IaaS, the problems of IaaS, the Cloud Security Alliance (CSA) threats, how the accountability and monitoring with logging systems assist in mitigating the risks associated with CSA threats, and an overview of the concepts of systematic provision of logging systems to support accountability in IaaS.

The remainder of the chapter is structured as follows. Section 2.1 provides the definition of cloud, and discusses its types. Section 2.2 provides an understanding of IaaS Cloud. It then describes both IaaS infrastructure and the components in the infrastructure, which will be referred to throughout this thesis. Then section 2.3 provides brief details of all CSA top threats to cloud computing.

This thesis mainly uses threat 1 as an example of how CSA threats effects both customers' and providers' security concerns. Thus, Section 2.4 gives more details of threat 1. Then, Section 2.5 discusses how threat 1 can have an impact on both customers and providers. Section 2.6 discusses accountability in the cloud to deal with CSA threats. This includes discussion of accountability and its properties, and of the association between the accountability and logging systems.

Section 2.7 discusses logging systems in IaaS in detail. Section 2.8 discusses related research concerning logging systems in IaaS. These works are referred to throughout this study. Section 2.9 discusses accountability, monitoring, and logging in IaaS. This includes a discussion of monitoring as a solution to the CSA threats, a logging system and its components in IaaS, and the differences between monitoring and logging systems.

After that, Section 2.10 discusses privacy and confidentiality concerns of customers and providers due to the log files, and how to deal with these concerns. Section 2.11 discusses how to systematically provide logging systems to support accountability in IaaS. Section 2.12 clarifies the meaning of an operating system, kernel, and file, which will be referred to throughout this thesis. Section 2.13 discusses TCB and logging systems. Finally, the chapter is briefly summarised and concluded in Section 2.14.

2.1 Definition of cloud

This section provides the definition of cloud, and discusses its various.

Cloud or Cloud computing is computing (e.g., executable program and data) at servers to other servers which are located elsewhere on the Internet [6]. [7] states that the cloud is a place where when we need technology, we can simply go to use it without any software installation, and we never pay for this technology if we do not use it. The cloud can be considered as a huge distributed system [5], which can provide services over the Internet. These services are applications hosted by data centres consisting of appropriate hardware and system software [5].

A cloud provides IT services to anyone who needs it in a pay-per-usage manner, like purchasing gas or electricity. These services provide for examples: computing or CPU power such as Amazon Web Services Elastic Cloud Computing or Amazon EC2 [17]; or web application development and hosting infrastructure such as Google AppEngine [56].

The CSA catalogues three types of cloud, in the Security Guidance Report [22] as follows: Infrastructure as a Service or IaaS, Platform as a Service or PaaS, and Software as a Service or SaaS. Note that this is a common subdivision. IaaS provides computer infrastructure with raw storage and networking to customers such as Amazon EC2 [17]. PaaS, for example Google App Engine [56], provides solution stack and computing platforms to assist in preparation of applications with the low cost and simplicity of purchasing and managing the fundamental hardware and software and providing hosting capabilities [22]. This includes supporting all processes of development and deployment of web applications and services fully available from the Internet. SaaS, for example Google Apps [57], is a software delivery model where the software and its associated data are hosted in the Internet and are accessed by users via, typically, browsers [22].

2.2 Infrastructure as a Service (IaaS) Cloud

This section provides further understanding of IaaS Cloud. It also describes both IaaS infrastructure and the components in the infrastructure. The infrastructure and the components will be referred to throughout this thesis.

IaaS consists of a cloud-based infrastructure to offer consumers raw computation resources such as storage and networking. These resources are billed using a pay-per-use cost model. IaaS provides a base on which to build PaaS or SaaS offerings, as pointed out in [22], [58]. Thus, comprehension of mitigating the risks associated with threats to IaaS may serve as guidelines to mitigate the risks to PaaS and SaaS. Section 2.2.1 discusses our version of IaaS architecture which is based on the Xen architecture. We represents IaaS architecture using Xen's base layers because Xen has been widely adopted in cloud computing platforms as argued by [59],[43].

2.2.1 IaaS Infrastructure

Each enterprise has its own cloud computing architecture, thus there is no standard architecture of the cloud [60]. This thesis focuses on IaaS cloud. An IaaS architecture (Figure 2.1) in this thesis is adapted mainly from papers [22; 39; 43; 46; 58; 61; 62] that discuss this phenomenon. Our version of an IaaS architecture will be referenced throughout this thesis for simplicity in explaining and discussing logging systems in IaaS cloud.

Figure 1.1 illustrates a public cloud computing environment in general. Alternatively, Figure 2.1 mainly presents the components inside the cloud infrastructure (Figure 1.1) as layers. These layers facilitate the investigation, understanding, and description of the locations where logging processes and log file components can be deployed in an IaaS infrastructure.

Our version of an IaaS architecture (Figure 2.1) is composed of a provider and a customer side. The provider side can be an organisation that offers services as rent-able virtual machines or VMs such as Amazon EC. A virtual machine is a software that can replicates a physical computer system which includes an operating system or OS, a main memory, disks, networking, etc. The customer side can be a person or an organisation that can remotely access these VMs via the Internet. Our version of IaaS architecture is based on the Xen architecture.

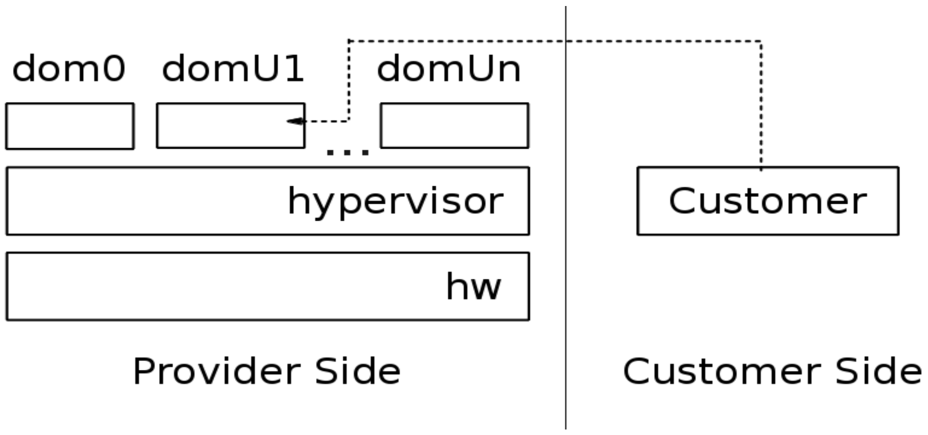


Figure 2.1: The IaaS architecture.

The main components of an IaaS architecture (Figure 2.1) are *hw*, hypervisor, dom0, and a number of domUs including domU1 to domUn. This architecture is a representation of one IaaS cloud physical machine. Real world IaaS cloud has a number of physical machines which are connected by the provider's network infrastructure. The components' in the architecture are described and listed below.

- ***hw***

  We use '*hw*' as an abbreviation for hardware which works as a physical machine that hosts a hypervisor, and all guest OSs. It is managed, maintained and owned by the provider who uses it to store and process customers' data.

- **A hypervisor**

  It is a layer of software running directly on *hw* and allows the *hw* to run multiple guest OSs at the same time [52]. It is an interface for all hardware requests from the guest OSs.

- **Dom0**

  It is a privileged domain guest OS that is launched by the hypervisor during system boot. Dom0 can be considered as a virtual machine management system and owned by a provider. Each physical machine has a dom0 that is responsible for the administration of all domUs in that particular machine. Thus, the dom0 directly accesses the *hw* and manages domUs. Note that, the term dom0 is from the term 'domain-0' which is from Xen's terminology

- **DomU**

  It is an unprivileged domain guest OS that runs on top of the hypervisor, but has no direct access to the *hw*. DomU is a VM which customers can rent it from a provider. Thus, it is virtually owned by a customer, and operates independently in the system. However, a domU can also be managed (e.g., launched, paused, or terminated) by a dom0. The term domU is from the term 'unprivileged domain' in Xen's terminology.

2.3 The Seven CSA Top Threats to IaaS Cloud Computing

This section provides brief details of all the Cloud Security Alliance (CSA) top threats to cloud computing.

One of CSA's well known reports in the cloud security area is *Top Threats to Cloud Computing, version 1.0* [23] which provides guidelines of the IaaS problems. This thesis intends to mitigate the risks associated with these threats. CSA [63] is a non-profit organisation with a mission to promote the use of best practice for providing security assurance within Cloud Computing, and to provide education on the uses of Cloud Computing to help secure all other forms of computing. CSA is led by a broad coalition of industry practitioners, corporations, associations and other key stakeholders.

All the threats can damage the trust of consumers in a cloud or IaaS environment, which is extremely important for its continued proliferation. Lack of trust in cloud slows wider deployment of cloud services [64]. For example, [23] argues that threat number five which is data loss or leakage and will be fully discussed below can have impact on customer morale and trust. Additionally, [65] also states that threat number three which is malicious insiders and will be fully discussed below also causes lack of trust of customers.

In the CSA report [23], these threats are for the security of IaaS, PaaS and SaaS. However, the threats discussed in the lists below are from the perspective of IaaS security.

- The first is *abuse and nefarious use of cloud computing.*

  For example, people with bad intentions can register legitimately to rent domUs, and later engage in malicious activities such as spamming.

- The second is *insecure application programming interfaces* (APIs) that are offered by providers.

  Customers have to use the APIs to interact with their rented domUs. If the APIs are insecure by design, they can be the cause of security issues such as confidentiality regarding customers' data in such domUs.

- The third threat is *malicious insiders* who usually have full privilege over dom0 in the IaaS environment.

  Thus, they may exploit customers' confidential data in domUs.

- The forth is *shared technology vulnerabilities.*

  For example, an owner of a domU may gain unauthorised access to the data of other domUs.

- The fifth threat is *data loss or leakage.*

  Customers' data loss may occur due to deletion of records without a backup of the original data in domUs.

- The sixth one is *account, service and traffic hijacking.*

Attackers re-use customers' credentials and passwords, which can be stolen through phishing and fraud. Thus, they can spy on customers' activities in domUs.

- The last threat is *unknown risk profile*.

  It is a fact that a customer does not know important information (such as who is sharing her infrastructure) that can be used to predict possible security risks of her domU.

As a result, these threats can be the cause of a lack of trust from the customers' points of view. Hence, we intend to study how accountability, monitoring, and logging approaches could assist in mitigating the risks associated with these threats as discussed in Section 2.6.

2.4 Threat 1 in detail

This thesis mainly uses threat 1 as an example of how CSA threats affect both customers' and providers' security concerns. Thus, this section gives more details of threat 1. Again, threat 1 is *the abuse and nefarious use of cloud computing*, which can occur when people misuse domUs such as spamming activities. 84% of respondents from two CSA survey reports of top threats to cloud computing [25; 26] are concerned that threat 1 is still relevant. There are many forms of threat 1. For example, the malware domain list website [66] provides a malware domain list. One can type in an EC2 keyword in the search tool of this site. It can be seen that there has recently been (2012/September/17) mis-usage of Amazon EC2 such as hosting Fake Flash page.

[67] states that the Zeus bot is a new type of cyber crime that uses EC2 as a command and control server. He also arguers that ScanSafe [68] found 80 unique malware incidents involving EC2. [69] argues that spammers and malware authors will continue to make a home in Amazon's EC2 service. [70] argues that there are malicious customers in the cloud provider's infrastructure. New forms of threat 1 have emerged, such as Pirate Bay which uses cloud severs to store their contents [71]. Thus, this thesis aims to provide systematic support for accountability in the cloud to mitigate the risks associated with this threat.

2.5 The effects of Threat 1 on Cloud Customers and Providers

This section discusses how threat 1 can have an impact on both customers and providers.

Section 5.2 will fully discuss the importance of critical files in domUs. These files can be any file type, such as text, executable, or database files. They are the customers' asset and valuable for their businesses. This thesis makes the case that threat 1 can cause the compromising of domUs and also of dom0. Consequently, this compromising can lead to undesired access to, or loss or leakage of, customers' critical files such as business databases. As a result, threat 1 can have an impact on both customers' and providers' companies. This can be in the form of the companies' brand damage, or productivity losses.

Some forms of attacks associated with threat 1 such as criminals using domUs to attack customer domUs or provider dom0 can be critical. This is because they can eventually cause the CSA threat 5 which is data loss or leakage. It is extremely difficult to recover the organisation's reputation or the lost data [72]. Threat 5 has been ranked as top in order of severity by the recent CSA reports [25; 26]. The CSA state the severity of threat 5 as:

"it can have a devastating impact on a business. Beyond the damage to one's brand and reputation, a loss could significantly impact employee, partner, and customer morale and trust. Loss of core intellectual property could have competitive and financial implications. Worse still, depending upon the data that is lost or leaked, there might be compliance violations and legal ramifications" [23].

2.6 Accountability in the cloud to mitigate risks associated with CSA threats

This section discusses accountability in the cloud as a method to deal with CSA threats. This includes a discussion of accountability and its properties (Section 2.6.1), and of the association between accountability and logging systems (Section 2.6.2).

2.6.1 Accountability

This subsection comprises a discussion of accountability and its properties. Logging systems can provide evidence to support accountability for an IaaS cloud, which helps us in mitigating the risks associated with the threats. Macmillan's dictionary [73] defines accountability as "a situation in which ***(1) people*** know ***(2) who*** is responsible for ***(3) something*** and can ask them to explain ***(4) its state or quality***".

Thus, this thesis views accountability in the cloud as "a situation in which ***(1') an auditor*** knows ***(2') a suspicious person*** is responsible for ***(3') incidents that were already occurred and that are causes of a CSA threat or threats*** and can provides ***(4') evidence*** to explain these incidents (that have already occurred).

An auditor (1') can be a provider, customer, or trusted third party auditor. A suspicious person (2') can be anybody who maliciously conducts activities that relate to a CSA threat or threats. An example of the incidents (3') can be a suspicious person accesses customer's critical files, after he takes control of the dom0, then over all domUs [74]. Evidence (4') refers to log files. An auditor's explanation of incidents (that are already occurred) needs evidence which is log files. These files can be used to explain the occurred incidents.

Accountability in the cloud properties [8] proposes four features for accountable distributed systems, as listed below. The cloud can be considered as a huge distributed system [5]. Thus, the accountable cloud should have these features.

- The first is *identities* which designates each action (such as the transmission of a message) as undeniably linked to the node that performed it.
- The second is *secure record* which means the system maintains a record of past actions such that nodes cannot secretly omit, falsify, or tamper with their entries.

- The third is *auditing* which is where the record can be inspected for signs of faults.
- The last is *evidence* which is when an auditor detects a fault, it can obtain evidence of the fault that can be independently verified by a third party.

[9] also propose the same idea to mitigate the risks associated with the problems of the cloud. They suggest that cloud customers have to be able to audit their data processing in the cloud. Thus, the customers can ensure that their data is not abused or leaked. If the data is abused or leaked, the proof of this fault can be obtained. Then [9] introduced a 'trusted monitor' to produce proof of compliance for the data owners. This proof of compliance can indicate that customers' data has not been manipulated without the access policies.

2.6.2 Accountability with logging systems

This subsection discusses the association between the accountability and logging systems. In the cloud, works discussed above that relate to the trusted monitor [9] and the accountable cloud [8] focus on dealing with the problems of the cloud using the same solution which is accountability. One of the signification mechanisms to enable the accountability properties above can be logging systems. To enable the ascertainment of evidence, log files are required as records of the past actions in the cloud servers. The following lists are examples of the benefits of logging systems which provide log data.

- [75] also states that log data is critically significant for analysing and replicating events that occur in cloud computing. The data also enables troubleshooting, fault diagnosis, operation audits, performance monitoring, resource robustness monitoring, detection and protection against intrusions, and a host of other usages [75].
- For accountability, trust, and security in cloud computing, preventive controls or detective controls can be trust components [45]. The former is used to mitigate the presence of an action from taking place at all. The latter is used to identify the occurrence of a privacy or security risk that

violates policies and procedures. Logs are one of examples of the detective controls which act as a psychological obstruction to go against policies or procedures in the cloud [27]. The approach also serves as a record for post-mortem investigations should any non-compliance occur [27].

Thus, Section 2.7 discusses logging systems and their related work in the cloud in depth.

2.7 Logging systems

This section discusses logging systems in IaaS in detail.

2.7.1 Logging systems

As discussed in Section 1.1, a logging system in IaaS is a system that resides in the provider's cloud infrastructure, and that can collect and store log information of the infrastructure's components such as domU. A logging system is composed of logging processes and log files.

A logging process performs logging tasks, whereas log files are used for storing contents produced by the logging processes. Each logging process has its own task(s). For example, one main task of a logging process called Flogger [30] is intercepting the file operations of a virtual machine or VM for the purposes of accountability in the cloud. It manages to store such intercepted data as log files.

The logging system's main components which include logging processes and log files can be distributed across domUs, the dom0, and the hypervisor. For example, the logging processes may be only: in a hypervisor as in [44]; in dom0 as in [1; 39]; in dom0 and domU as in [30; 43]; or in only domU as in [76].

Recent works [8; 9] give an outline of accountability/monitoring-related solutions for the cloud. These works involve the logging processes and the log files, which will be in the cloud machines. However, they do not discuss where exactly they could be in the machines (which involve visualisation of infrastructure and more than one party), and who will manage them and why.

Other works that involve design and implementation of the logging solutions which could be applied in the cloud include [27; 39; 43; 44]. These works focus on monitoring and logging tasks. Ideally, a logging system can be analysed, with respect to its features and achievable goals, before deployment. However, these authors do not give completed concerns about the security analysis of their logging systems themselves, which include the logging processes and the log files before deploying the systems to the real world IaaS.

Moreover, [58] propose the security model of IaaS. The model includes the security model side that comprises of two entities. One of these entities is the security policy monitoring and auditing, which can track the system life cycle. This entity should involve logging systems. However, they do not provide implementation and examples of the model, and do not focus on dealing with CSA threats.

2.8 Related work concerning logging systems in IaaS

This section discusses related work concerning logging systems in IaaS. These works are referred to throughout this study. We investigated previous research that can be considered as work that could be applied in a logging approach in IaaS.

Firstly, TrustCloud [27] is a framework proposed by HP to address the lack of trust in the cloud. Its most important layer is the system layer. It is a foundation to build other layers, and deploys Flogger [30] as its core. Flogger is an interceptor that can be placed in the dom0 kernel or domU kernel to intercept the file and network operations of that domain.

Secondly, accountable virtual machines or AVMs [44] are virtual machines that can detect mis-behaviors of online gaming servers by using a modified hypervisor or VMware to record all messages sent and received by an untrusted server. The main process of this work is a logger that resides in the hypervisor to record incoming and outgoing network packets of domUs.

Thirdly, PASSXen [43] is an approach to collect the system-level provenance of domUs that run under Xen. PASSXen's interceptor in domU kernel tracks and collects the creation, access, and destruction of processes and files for this domU. Fourthly, [1] proposes a network monitoring application which identifies which process inside a Windows domU is responsible for malicious network traffic leaving this domU.

Then, it ([39]) is a demo monitoring program in dom0 that outputs all file/directory creation/removals happening in domU's */root* directory. Then, Lares [77] can monitor domU behaviours. It may be possible to modify this system to produce log files. This requires insertion of hooks at runtime into a domU.

Lastly, to address the problem of cloud customers that fear loss of control of their own data, particularly financial and health data, that are in providers' machines, JAR logging [76] proposes log files to keep track of the actual usage of the customers' data in the cloud. It is very useful for customers to check the transactions of their files. This work compresses a user file (such as an image file) along with any policies (such as access control policies and logging policies) and the logging mechanism (includes an automated and authenticated logging mechanism) as a Java ARchive or JAR file. Any access to the user file will trigger the logging of an automated and authenticated logging mechanism. For example, suppose that a cloud service provider with ID Kronos, located in the USA, reads a customer image file at 4:52 pm on May 20, 2011. The corresponding log record is Kronos, View, 2011-05-20 16:52:30, USA.

2.9 Accountability, Monitoring, and Logging in IaaS

Accountability, monitoring, and logging are associated. We argues that one important mechanism for accountability is monitoring which can deploy a logging system as its core. This section discusses accountability, monitoring, and logging in IaaS below. This includes the discussion of monitoring as a solution to the CSA hreats, a logging system and its components in IaaS, and the differences between monitoring and logging systems.

- **Monitoring as a Solution to the CSA Threats**

  In [23], CSA states that a monitoring approach is one of the remediations in its list of solutions for each threat. It also argues that monitoring can mitigate the risks associated with threats 1, 3, 4, 6, and 7. In [27], HP also summarises that some log files such as histories of file access on the provider side can aid the providers and the consumers to mitigate CSA threats 1, 2, 3, 5 and 7. However, this thesis considers a monitoring system to be a system that can monitor which activities take place inside dom0 and/or domUs, and its main subsystem is a logging system.

- **A logging system and its components in IaaS**

  The basic idea of a logging system is to use logging processes to record particular information, such as in [44] the logging processes record incoming and outgoing network packets of domUs. Then they store such information as log files to be used for particular purposes such as evidence for identifying malicious network traffic in the domUs. This thesis does not discuss real-time elements of monitoring such as triggers, because these elements should be fundamental for a monitoring system, and should always be considered and implemented before deploying the system. We instead focus on the security analysis of logging processes and log files, which are critical components in a logging system.

- **Differences between monitoring and logging systems**

  However, we need to differentiate works for monitoring and logging. This is because the work for monitoring does not need to have log files, whereas logging work have to have log files. The locations of log files is an indicator of the security and reliability of a logging system. This thesis defines monitoring works as: monitoring can monitor domU's activities, then may analyse these activities, then may detect malicious activities, then prevent them. Examples of monitoring works are CloudSec [78], libVMI [79], a demo monitoring program in dom0 [39], and a network monitoring application [1].

  In contrast, a logging system is a regular logging such as CCTV, without analysing the activities while doing logging. The result of the logging system is log files which will be used later, if malicious incidents occur.

  Thus, the main difference is the monitoring systems do not produce log files, whereas logging systems do. Thus, without consideration of logging files, it may be easier to develop monitoring systems compared to logging systems that need to consider the locations of log files which will be permanently or temporally stored in media in those locations. This is because the locations can be one of the indicators of the security of the log files themselves. As a result, to develop a logging system is more difficult than monitoring system. Thus, this thesis considers only logging systems that have log files. All monitoring systems in this thesis do not have log files. However, all monitoring work can be modified to produce log files, but they need to consider the locations of log files. Table 2.1 again summarises monitoring and logging work in the cloud.

Monitoring	CloudSec [78], libVMI [79], a demo monitoring program in dom0 [39], and a network monitoring application [1]
Logging	PASSXen [43] (do not discuss where is the log files will be in the IaaS architecture), HP Flogger [30] (the log files are in dom0), AVMs [44] (the log files are in dom0)

Table 2.1: Monitoring and Logging work in the IaaS

2.10 Privacy and Confidentiality Concerns of Customers and Providers due to the Log Files

This section discusses the privacy and confidentiality concerns of customers and providers due to the log files, and how to deal with these concerns.

Log files discussed in this thesis can be detailed records of activities of processes and files in a customer VM. For auditing purposes, the records can be investigated by a third party. Thus, this information may disclose customer business activities, or malicious activities inside a provider's cloud infrastructure. This leads to privacy and confidentiality issues for both customers and providers. However, [8; 27] argue that privacy issues are manageable. For example, to manage the privacy issues, [8] states that it is important to consider what is being recorded, and who can access this recorded information; thus, logging system management should return the recorded information at different levels of detail, depending on who needs it. To address the privacy and confidentiality issues, this may lead to further research such as balancing the privacy and its usage such as [80]. This is out of the scope of this thesis.

2.11 A plan for Systematic provision of logging systems to support accountability in IaaS

This section discusses how to systematically provide logging systems to support accountability in IaaS.

The generic framework we aim to design (stated in Objective 2 and 4, Section 1.3) is the same as a plan to systematically build up logging systems to support for accountability in the cloud. In an IaaS environment, this framework should be used to encompass all possible instantiations of logging system, according to the system owner's requirements.

Systematic logging system development with a plan The logging system development also needs a plan or should be systemic. A logging system is a software system. In IaaS, what makes it special is that it will be in the virtualisation environment instead of in local standalone machines in private organisations.

Systematic logging system development with the concerns of security issues of log files and logging processes As discussed in Section 2.9 we need to consider where the log files that are produced from logging systems will be stored in IaaS architecture. This is because the locations of the log files can affect their security such as the integrity. The next chapters will fully discuss the security issues of log files, as well as of logging processes.

2.12 Overview of Operating Systems, Kernels, and Files

This section clarifies the meaning of an operating system, kernel, and file, which will be referred to throughout this thesis.

This overview is derived from [2]. For a computer machine, the term system refers to an operating system or OS and all the applications running on top of this OS. An operating system is considered as the parts of the system, which are responsible for basic use and administration of the machine. These parts are the kernel and device drivers, boot loader, command shell or other user interface, and basic file and system utilities. The kernel is one of the OS's parts that is responsible for basic use and administration of a computer machine, such as the Linux kernel for Linux OS.

The kernel typically resides in an elevated system state compared to normal user applications. This includes a protected memory space and full access to the hardware. This system state and memory space is collectively referred to as kernel-space. In opposition, user applications are executed in the user-space. They see a subset of the machine's available resources and can perform certain system functions, directly access hardware, access memory outside of that allotted them by the kernel, or otherwise misbehave. When executing kernel code, the system is in the kernel-space executing in kernel mode. When running a regular process, the system is in the user-space executing in user mode.

A filesystem is a hierarchical storage of data adhering to a specific structure. A filesystem contains files, directories, and associated control information. A file is an ordered string of bytes. Each file is assigned a human-readable name for identification by both the system and the user. Typical file operations are read, write, create, and delete.

2.13 Trusted Computing Base or TCB

This section discusses TCB and logging systems.

[40] points out that an OS is difficult to analyse because of its size and complexity. He also argues that there is too much TCB when deploying an application in an OS such as a domU that is running on top of a hypervisor. [81] argue that an OS code changes rapidly over time. The changes may increase the size of the OS and its complexity, and as a result, its TCB.

Thus, the TCB can be a very important aspect of security, in order to propose logging systems in the cloud. For example, the proposed systems in [38; 39; 40; 41] (that can be considered as solutions to mitigate the risks associated with the cloud problem) are concerned with reducing the TCB size along with their proposed systems. Recent work that extensively relates to a secure cloud computing environment such as that of [42] also focuses on reducing the TCB in their proposed architecture.

2.14 Conclusions

The main contribution of this chapter was to present a background and literature review for the thesis. This chapter discussed the definitions of cloud computing and IaaS, the problems of IaaS, the CSA threats, how the accountability and monitoring with logging systems assists in mitigating the risks associated with CSA threats, and an overview of the concepts of systematic provision of logging systems to support accountability in IaaS.

Chapter 3

Generic Logging Components for Infrastructure as a Service (IaaS) Cloud

This chapter addresses Gap 1 (c) which is the lack of security analysis of the logging systems themselves before the deployment of the systems in to the IaaS real world productions, as discussed in Section 1.2. The previous chapter provided the holistic representation of the environment of logging systems in the cloud. This representation is utilised as the basis for the design of a generic framework of logging solutions to mitigate the risks associated with CSA threats (Objective 2). We call this framework **generic logging components for infrastructure as a service or IaaS cloud**.

Thus, the main contribution of this chapter is these generic logging components. To facilitate systematic support for accountability in the cloud, these generic logging components provide ways to build logging systems. The value of these generic logging components is to encompass all possible instantiations of logging solutions for IaaS cloud, and to provide a clear view of all components that relate to logging systems in IaaS. This view provides a basis for the analysis of logging systems' security before deployment. Thus, these generic logging components enable logging systems to be appropriately designed or manipulated by participating cloud parties such as a provider, customer, or auditor. As result, this enhances systematic support for accountability in the could.

The remainder of the chapter is structured as follows. Section 3.1 discusses why we need generic logging components for accountability in IaaS. Section 3.2 discusses importance of security analysis of logging systems in IaaS. Then, Section 3.3 proposes generic logging components of IaaS cloud. This section provides the details of the generic logging components and each component. Then the section discusses how these components assist in analysing of the security of logging systems. The analysis include integrity and privacy issues of a logging system.

Section 3.4 provides two case studies of how to use the generic logging components, and discussion relates to the case studies. This section includes a case study of mapping HP Flogger on the generic logging components for the purposes of security analysis include integrity and privacy issues of Flogger system, a case study of an identification of the appropriate logging system based on the generic logging components to mitigate the risks associated with CSA threat 1, discussion of possible solutions of the security issue of log files as F3 in the logging system in the spam case study, and discussion of to detect other forms of attacks of threat 1 using logging solutions.

Section 3.5 provides evaluation and discussion of the generic logging components. The section includes an evaluation of the generic logging components against how these components satisfy their goals, and many other discussions of the generic logging components, which affect many aspects of logging systems in IaaS, as well as PaaS such as applying the generic logging components to mitigate risks associated with CSA threats for PaaS, and a TCB size measurement of a logging system in the cloud based on the generic logging components. Finally, the chapter is briefly summarized and concluded in Section 3.6.

3.1 The need for generic logging components for accountability in IaaS

This section discusses why we need generic logging components for the accountability in IaaS.

We use the term generic logging components to indicate that we consider the union of all possible logging architectures that could be instantiated. In that sense, the generic logging components by themselves are not meant to be implemented, but offer the building blocks from which one can choose to create an actual logging solution. Moreover, with respect to faster system development, the generic logging components are flexible and implementation independent, and provides reuse-ability. Importantly, it can be used as a tool for security analysis of logging systems in IaaS, as will be discussed in Section 3.2. It could be a starting point of building accountability systems to address lack of trust in an IaaS.

3.2 The need for security analysis of logging systems in IaaS

This section discusses importance of security analysis of logging systems in IaaS.

[82] state that an important obstacle to users in adapting into cloud computing is security and privacy issues. They also argue that to realise the prosperity in cloud computing literature, those issues need to be addressed beforehand. This perspective can be applied to the logging process as well: security and privacy issues of the logging systems have to be resolved before using them.

The security analysis of logging systems themselves is very important because the logging processes and log files are critical components and so they need to satisfy a set of security properties such as integrity and privacy. Hence, without the security analysis of logging systems, it could be difficult to effectively and efficiently build and deploy logging systems that can satisfy those security properties.

An IaaS environment, which involves virtualisation and more than one party, makes the security analysis of logging systems more complicated than one within a private organisation. Haeberlen [8] pointed out that one of the research challenges of the accountable cloud is that it needs to have mechanisms to allow legacy users to access the logging machine which deploys logging processes, but not to maintain the log files. Crosby and Wallach [83] agreed that ensuring the integrity of the log files is a critical part of a larger system.

This thesis considers the security analysis of logging systems as a significant criterion to establish the goals of the generic logging components as will be discussed in Section 3.3.1. Then, this chapter focuses on where the logging processes and log files can precisely be inside the IaaS architecture. This is because the locations of these components in the IaaS can directly and significantly affect the security concerns of these components.

3.3 Generic logging components of IaaS cloud

This section proposes generic logging components of IaaS cloud. It provides the details of the generic logging components in Subsection 3.3.1, the details of each component in Subsection 3.3.2, and a discussion of how the generic logging components assist in the analysis of security of logging systems in Subsection 3.3.3. The analysis include the analysis of integrity and of privacy issues of a logging system. In this chapter, the terms 'generic logging components' and 'logging components' are used interchangeably.

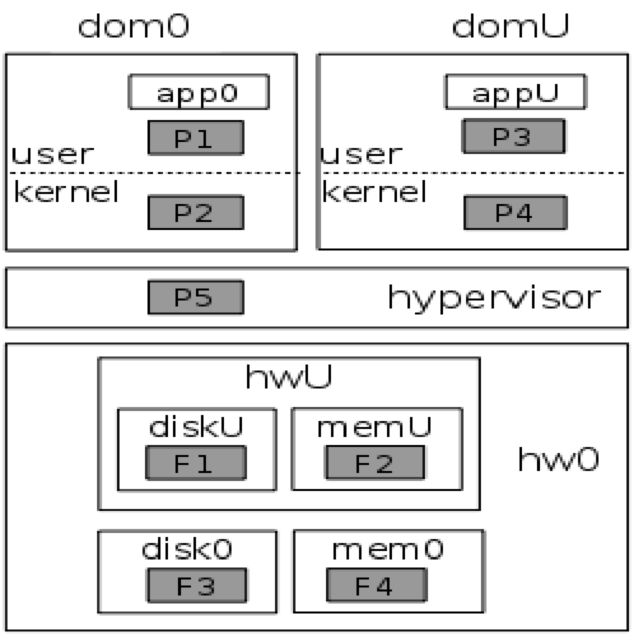


Figure 3.1: The overall view of generic logging components: logging process or Px (P1 to P5), and log files or Fy (F1 to F4).

We investigate logging process and log file components of each work from previous research concerned with logging systems from the related work section or Section 2.8. Then we consider the possible locations of logging processes and log file components in the IaaS architecture.

Section 2.3 provides details of the CSA threats. Section 3.2 states that the security analysis of logging systems is critical before they are deployed into the real world IaaS. Below are two goals of the generic logging components to deal with the threats, regarding the security analysis of logging systems.

1. The generic logging components can be used to instantiate new logging system architecture to mitigate the risks associated with all CSA threats in the IaaS environment.

2. The security of the newly built logging system architecture, that is based on the generic logging components, can be systematically analysed before deployment.

3.3.1 Details of the generic logging components

This section discusses the details of the generic logging components.

Figure 3.1 is the generic logging components. These components are divided into three sets of components: the IaaS set of components, the logging processes' set of components and the log file's set of components. All white boxes in Figure 3.1 are the IaaS set of components. They are hw0, hypervisor, hwU, dom0, domU, app0, appU, disk0, diskU, mem0, and memU. The shaded boxes in dom0, domU, and hypervisor in Figure 3.1 are the logging processes' set of components or Px. They are P1 to P5. The log file's set of components or Fy is composed of F1 to F4, see the shaded boxes in hw0 in Figure 3.1.

Discussion of Locations of Logging processes/Px and log files/Fy Px can have different functions and can be deployed in different locations in the IaaS layers when dealing with different CSA threats. Figure 3.1 and Tables 3.1 and 3.2 show that there are five locations for Px. This can be in dom0 user level as P1, dom0 kernel level as P2, domU user level as P3, domU kernel level as P4, or hypervisor as P5.

Px	Description	Examples and Functions	Locations in IaaS environment
P1	1. a dom0 user level process or 2. special dom0 libraries for logging purposes	-in PASSXen[43]: *Waldo* (to read provenance records from a log, store these records in a database, index them, and access the database for querying), or *Coordinator-like* (to periodically consume temporary logs from mem0, and perform other tasks by communicating with other dom0 kernel level Px or dom0 user level Px) -in libVMI [79]: *Libraries* (to be called, for example, by app0 to read memU for introspection purposes)	in dom0 user level
P2	a dom0 kernel level process	-in HP Floggers System [30]: *Flogger* (to intercept dom0 file and network operations, then store such intercepted data into disk0) -in PASSXen [43]: *Analyser* (to process the provenance records)	in dom0 kernel level
P3	1. a domU user level process or 2. special domU libraries for logging purposes	-in a logging system of [76]: *Logging mechanism* (to record the actual usage of data files that reside in diskU) N/A	in domU user level
P4	a domU kernel level process	-in HP Floggers System [30]: *Flogger* (to intercept domU file and network operations, then temporarily store such intercepted data into diskU in case the data are too big to be stored in memU) -in PASSXen [43]: *Interceptor* in domU (to intercept domU system calls, and temporarily store such intercepted records in a mem0)	in domU kernel level
P5	a process inside a hypervisor	-in AVMs system [44]: *Logger* (to record information about incoming and outgoing network packets of domU)	in a hypervisor

Table 3.1: The Logging Process set of components (the shaded boxes in dom0, domU, and hypervisor in Figure 3.1)

Component	Description	Examples and Functions	Locations in IaaS environment
F1	temporary or permanent log data in diskU	-in Floggers System [30]: a domU HP Flogger (interceptor/P4) could temporarily store the domU intercepted data as *F1* in diskU (in case the data are too big to be stored in memU) for further processing	in diskU
F2	temporary log data in memU	-in HP Floggers System [30]: the same as the example of F1, but Flogger/P4 could eventually and temporarily store the log data as *F2* in memU for further processing	in memU
F3	temporary or permanent log data in disk0	-in HP Floggers System [30]: Flogger/P2 can permanently store the intercepted data (as *F3*) in disk0 -in AVMs system [44]: a Logger (P5) permanently stores the domU intercepted network data (as *F3*) in disk0	in disk0
F4	temporary log data in mem0	-in PASSXen [43]: domU Interceptor (P4) temporarily stores intercepted domU system calls data (as *F4*) in mem0	in mem0

Table 3.2: The log file set of components (the shaded boxes in hw0 in Figure 3.1)

In the generic logging components, we place Fy only in the locations of either a primary or main memory which includes mem0 or memU, or a secondary memory that includes disk0 or diskU. Other locations including dom0 kernel and user level space, domU kernel and user level space, or in a hypervisor are for locating or running Px, appU, and app0, rather than storing Fy.

Thus, there are four locations for Fy: diskU, memU, mem0, and disk0. Fy can be a temporary Fy which is temporarily stored in either diskU as F1, memU as F2, disk0 as F3, and mem0 as F4 for further processing then it will be removed. For example, in HP Floggers System [30], a Flogger intercepts domU file and network operations, then temporarily store such intercepted data into diskU as F1 in case the data are too big to be stored in memU; or in PASSXen [43], an interceptor intercepts domU system calls, and temporarily stores such intercepted records in a mem0 as F4.

Fy can also be a permanent Fy which is stored in diskU as F1 or disk0 as F3. For example, in an AVMs system [44], a logger as P5 permanently stores the domU intercepted network data in disk0 as F3. Locations of Px and Fy can be used as their security indicators such as privacy of Fy, as stated in Section 2.9 and 3.2 and as will be demonstrated and discussed in the case studies in Section 3.4. We consider mem0 and memU as components of the generic logging components because log data will eventually and temporarily be in these memory which affects the security analysis of such log data.

3.3.2 The details of each generic logging component

This subsection provides the details of each component in the generic logging components. The details of all components from all sets are listed below.

1. **The IaaS set of components**

   All components of this set are derived from the IaaS architecture (Figure 2.1). Note that a component that has its name ends with 0/zero (e.g., hw0) is a component that is physically owned and managed by a provider. A component that has its name ends with U (e.g., hwU) is a component that is virtually owned and managed by a customer. Hw0 (e.g., a PC or server) is the hw described in Section 2.2. It is a host machine for hypervisor, dom0, and all domUs. It is in the provider side infrastructure and managed and owned by a provider through dom0. It consists of a disk0 and mem0. Disk0 is a physical disk of the hw0 such as server's disks. It is managed by dom0 and physically in hw0. Mem0 is a main memory of the hw0 and physically in hw0 such as a server's main memory.

Hypervisor, for example Xen, is described in Section 2.2. It concurrently runs a dom0 and domUs, and is on top of hw0. In the generic logging components, system memory of dom0 and domU, which are discussed below has two parts: a user level space; and a kernel level space. The former is a set of memory locations in the system memory, that the user process such as appU runs, discussed below. The latter is the system memory where the kernel that is the main part of the OS runs [84], as discussed in Section 2.12.

Dom0 is described in Section 2.2. It consists of app0 and is on top of the hypervisor. An example of dom0 is a Fedora 16 Linux system. App0 is an application that runs inside the dom0 user level through a provider such as a logging-related application. DomU is described in Section 2.2. It consists of appU and hwU, and is on top of the hypervisor. An example of domU is a Fedora 16 Linux system.

AppU is an application that runs inside the domU user level through an owner of this domU. An example of appU is a mail application which is a *mail* command in Linux system. HwU is a virtual hardware that consists of diskU and memU. It is owned by domU's owners, and resides virtually in domU, but physically in hw0. DiskU is a virtual disk of domU. It is virtually in hwU, but physically in disk0/hw0. The last component of the IaaS set of components is memU. It is a virtual main memory of domU. It is virtually in hwU, and physically in mem0/hw0.

2. **The logging processes' set of components**

   The components of this set are P1 to P5 which are presented in Table 3.1 and Figure 3.1. We place only one Px in particular locations as a representative of multiple Px in each location. Hence, it is possible to have more than one Px in a particular location. For example, P2 in a dom0 kernel level could actually compose of more than one logging processes, depending on the design and implementation of an actual logging system that P2 belongs to. Any Px can also collaborate with another or others to achieve logging tasks.

Table 3.1 presents description, examples, functions, and locations in the IaaS environment of each component in this set. Note that this section refers to OS system calls. They are the interface to the OS kernel, and are used to request services offered by, and implemented in the kernel [85].

Each Px (Table 3.1) that is discussed in this chapter has different properties or abilities compared to others. The properties and abilities of each Px are discussed in Table 3.1. It is possible that each Px has the same properties and abilities as others. This depends on the design of logging systems. However, the discussion of Px in this chapter focuses on these P1 to P5 (Table 3.1). This is because we focus on the security analysis of logging systems, which is based on the security of their critical components (e.g., Px and log files) that can be distributed across an IaaS architecture.

Thus, instead of discussing all possible properties or abilities of each Px, we focus on the locations across an IaaS architecture where Px can be deployed such as in dom0 user level or in domU kernel level. The deployment location of a Px can assist in analysing security of this Px. Section 3.3.1 provides a full discussion of locations of P1 to P5, and Section 3.5.2.1 discusses other possible locations of these Px apart from their locations in the generic logging components.

3. **The log file's set of components**

   All components of this set are presented in Table 3.2. This set is composed of F1 to F4 which are all the shaded boxes in hw0 in Figure 3.1. The representative approach of Px can apply to Fy as well. Thus, it is possible to have more than one Fy in disk0. Table 3.2 presents description, examples, functions, and, locations in the IaaS environment of each component in this set. For these generic logging components, Fy can be stored in memory (mem0 or memU) and disks (disk0 or diskU). Section 3.3.1 provides a full discussion of locations F1 to F4, and Section 3.5.2.1 discusses other possible locations of these Fy apart from their locations in the generic logging components.

3.3.3 Discussion of how the generic logging components assist in analysing the security of logging systems

This subsection discusses how the generic logging components assist in analysing the security of logging systems. This includes analysis of integrity, and of privacy issues of a logging system.

To facilitate discussion of analysing the security of existing and our proposed logging systems in Sections 3.4 and Chapter 5 to 6, we discuss how the logging components assist in analysing the security of logging systems with integrity or privacy issues in this section (Section 3.3.3). Then, Section 3.5.2 will discuss many other aspects of the logging components.

The logging components can assist in analysing the security of logging systems in Iaas. Again, the logging components can be used to instantiate architecture of a logging system by choosing appropriate Px and Fy from the logging components to form the architecture based on the requirements of this logging system. Then, this logging system can be built up later. It should be built after its security analysis has been achieved, and security issues of the architecture/system have been resolved by all parties that are involved in this logging system.

The logging components can assist in analysing the security of logging systems. This is because locations of Px and Fy in the logging components can facilitate the analysis of the security of any logging system in IaaS when the architecture of this system is designed based on the logging components.

[86] argues that in the security of computing and communication systems area, security comprises three attributes: confidentiality, integrity, and availability. They also give the definition of each attribute as follows. Confidentiality is the absence of unauthorised disclosure of information. Integrity is the absence of improper system alterations. Availability is the readiness for correct service. Moreover, confidentiality relates to the broader concept of data privacy which is the

limitation of access to individuals' personal information [87].

We discuss only integrity and privacy attributes as an example of how the logging components assist in analysing the security of a logging system designed based on the logging components. From the logging components, there are nine locations that can be used to deploy Px and Fy. Five locations are for deploying Px: dom0 user level, dom0 kernel level, domU user level, domU kernel level, and in an hypervisor. The other four locations are for deploying Fy: diskU, memU, disk0, and mem0. These nine locations can be indicators of the integrity and privacy concerns relates to a logging system, as discussed in Section 3.3.3.1 and 3.3.3.2 below.

The architecture of any logging system that is designed based on the logging components can be used as a tool to analyse integrity and privacy issues which relate to this logging system. Thus, the logging components assist in analysing the security of logging systems in IaaS. In addition, when the architecture of any existing logging system in IaaS can be mapped on the logging components, this mapping architecture can be used as a tool to analyse the security of the system.

We consider the logging components as a tool to assist in analysing the security of any logging system. This is because the logging components can be a neutral reference model or tool for security analysis of logging systems in an IaaS. There is no united or general tool available to analyse the security of logging systems in an IaaS at the moment. Our logging components can be a candidate. The analysis of security of logging systems is an important aspect to build logging systems to support accountability in IaaS, as discussed in Section 1.1, 1.2, 2.9, and 3.2. Without the logging components it would not be easy to systemically build logging systems to support accountability in an IaaS.

3.3.3.1 Analysis of integrity issues of a logging system

From the logging components (Figure 3.1), when any logging system deploys Px in domU user level as P3 or in domU kernel level as P4, this may lead to integrity issues of this logging system. This is because a domU cannot be trusted from the providers' perspective to install security software (e.g., logging systems) inside it, as argued by [78]. This domU may tamper with the logging systems and alter the system's behaviour, whether by the domU owner or by an external attacker.

When any logging system deploys Px in the dom0 user level as P1 or in dom0 kernel level as P2, this may also lead to integrity issues of this logging system. This is because dom0 is physically owned and controlled by a provider, as described in Section 3.3.2. Therefore, the provider's employees may maliciously modify P1's code to produce contents of log files which benefit themselves. These employees are called malicious insiders which are serious issues as argued by [46; 88; 89].

3.3.3.2 Analysis of privacy issues of a logging system

When any logging system deploys Fy in disk0 as F3, this may lead to privacy issues. This is because malicious insiders may learn about customers' personal data, which is a privacy violation to a customer who owns this domU. Rocha and Correia [41] demonstrate how this kind of attack can obtain confidential data from a customer domU. [90] also agree that it is easier to steal customer's data if the thief is an insider. Thus, it is also easier for malicious insiders to tamper with or steal F3; this is because these files are in disk0 which is controlled by the insiders. This issue needs to be addressed before deploying F3.

3.4 Case studies of how to use the generic logging components

To present how the logging components work, this section provides two case studies of how to use the logging components, and the related discussion. This section includes: a case study of mapping HP Flogger on the logging components for the purposes of security analysis including integrity and privacy issues of the Flogger system in Subsection 3.4.1; a case study of an identification of the appropriate logging system based on the logging components to mitigate the risks associated with CSA threat 1 in Subsection 3.4.2; a discussion of possible solutions to the security issue of log files such as F3 in the logging system in the spam case study in Subsection 3.4.3; and a discussion of how to detect other forms of attacks of threat 1 using logging solutions in Subsection 3.4.4.

3.4.1 A Case Study of Mapping HP Flogger on the logging components

This subsection provides a case study of mapping HP Flogger on the logging components for the purposes of security analysis of the Flogger system. This includes analysis of the system's integrity issues in Section 3.3.3.1 and privacy issues in Section 3.3.3.2.

As stated earlier in Section 3.3.2, a logging system can have more than one Px in a particular location, depending on its design and implementation. The Flogger system [30] actually consists of more than one Px in its dom0 and domU kernels. We assume this mapping is for the purposes of security analysis of the Flogger system itself.

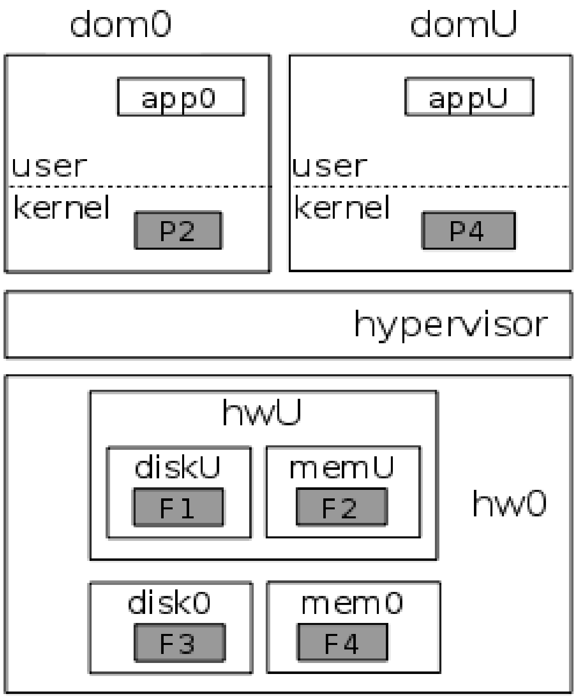


Figure 3.2: Mapping some HP Flogger Px and Fy on the logging components.

The Flogger system (Figure 3.2) deploys six critical components: two logging processes components including P2 and P4, and four log file components which are F1 to F4. P2 actually contains four components: a receiver, interceptor, consolidator, and writer. Whereas P4 contains two components: a sender and an interceptor.

The main task of the receiver is taking log data from the sender. The P2's interceptor intercepts dom0 file and network operations. The consolidator produces the final version of log data. The writer writes the final version of log data to disk0 as F3. The final version of log data must eventually and temporarily be in mem0 as F4. For P4 side, the sender transfers the log data from domU to the dom0's receiver. P4's interceptor intercepts domU file and network operations. It may temporarily store such intercepted data into diskU as F1, in case the data are too big to be stored in memU as F2.

Analysis the security of Flogger system can be achieved using the mapping (Figure 3.2), as discussed in the lists below.

- **Analysis of integrity issues of Flogger system**

  After mapping the Flogger (see Figure 3.2), the Flogger system deploys P4 at the domU kernel level, this may lead to integrity issues of this logging system. This is because this domU owner may tamper with P4, as discussed in Section 3.3.3.1.

- **Analysis of privacy issues with Flogger system**

  From the mapping, one can analyse the privacy of F3 because it is in disk0 which is a part of hw0/dom0 (as described in Section 3.3.2) and is physically owned by a provider. Therefore, malicious insiders may learn about, or alter F3, which is a privacy violation to a customer, as discussed in Section 3.3.3.2.

3.4.2 A Case Study of an Identification of the Appropriate Logging System Based on the Logging Components to Mitigate the Risks Associated with CSA Threat 1

This subsection is another case study to demonstrate how to use the logging components. This case study is an identification of the appropriate logging system based on the logging components to mitigate the risks associated with CSA threat 1. This subsection includes (for this case study): design and implementation of the logging system, how to run the experiment, the result, and the analysis of security of the logging system.

In order to satisfy the goals of the logging components, this case study demonstrates how to use the logging components to instantiate the logging system architecture, and how to analyse the security of this new built architecture.

3.4.2.1 Design and Implementation of the Logging System

This logging system architecture is designed and based on the logging components. Thus, we can choose some components from the logging components (Figure 3.1 and Section 3.2-3.3) to build up the logging system. At the time of the experiment, we found libVMI [79] as the only available tool that can achieve the goal of the case study. This is an introspection library to read memory of VMs or domUs from dom0. We deployed libVMI as P1 in dom0 user level to read the memU of this spamming domU.

Therefore, we reuse libVMI as P1, then we choose F3 as log files to build the logging system architecture. As a result, Figure 3.3 shows the overview of the

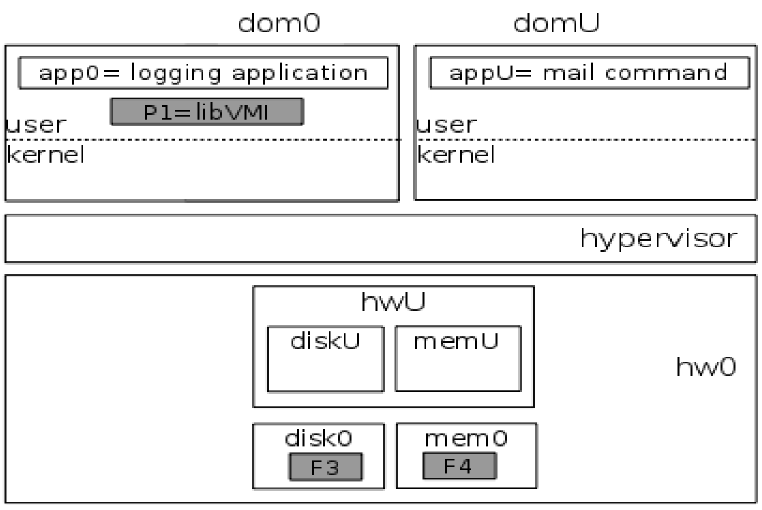
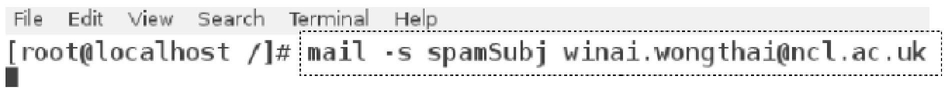


Figure 3.3: The logging system architecture to mitigate the risks associated with threat 1.

newly designed logging system architecture according to this case study's goal. It is quite clear that the new architecture is composed of only P1 or libVMI, F3 or log files, F4 or temporary log data, and app0 or a logging application.

This case study simulates spam activities by assuming that spammers rent a Linux VM or domU from an IaaS provider. They then use appU (the *mail* command, see Figure 3.4 in the dot-line box) to send a spam email to a vic-



```
[root@localhost /]# mail -s spamSubj winai.wongthai@ncl.ac.uk
█
```

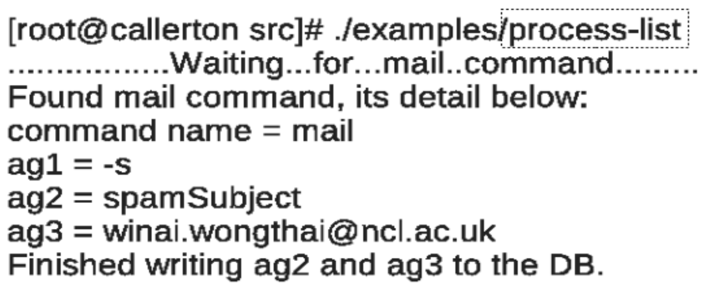
Figure 3.4: Spam activities performed by appU (*mail* command) in domU.

tim. The command (we call *c1*) used as simulation of spam activities is: *mail -s spamSubj winai.wongthai@ncl.ac.uk*. The *mail* command in *c1* has three arguments. They are *ag1* which is -s, *ag2* which is spamSubj, and *ag3* which is winai.wongthai@ncl.ac.uk. It sends string 'spamSubj' as an e-mail subject to the email address winai.wongthai@ncl.ac.uk. In order to capture *ag2* and *ag3*, we assume that the logger application knows the name of the *mail* command in *c1*.

The case study's goal is to enable a logging application to capture the arguments of *c1* including *ag2* and *ag3*, and then writes the captured *ag2* and *ag3* to a database as the log files as F3. These log files can be used by an auditor, a trusted third party, or a provider as the evidence to afterwards identify spam activities in this rented domU later. We assume that *ag2* and *ag3* are sufficient to identify these spam activities.

3.4.2.2 Running the Experiment in this Case Study

We run the logging application inside the dom0 user level, see *process-list* command line in the dot-line box in Figure 3.5. It keeps checking memU until *c1* is performed. When *c1* is performed as shown in Figure 3.4, then the logging application extracts the arguments of *c1*, which are *ag2* and *ag3* or the second and third lines from the bottom in Figure 3.5. The application then writes both extracted arguments to F3.



```
[root@callerton src]# ./examples/process-list
................Waiting...for...mail..command........
Found mail command, its detail below:
command name = mail
ag1 = -s
ag2 = spamSubject
ag3 = winai.wongthai@ncl.ac.uk
Finished writing ag2 and ag3 to the DB.
```

Figure 3.5: The logging application in dom0 extracts arguments of c1 (*ag2* and *ag3*) as evidence for identifying spam activities.

3.4.2.3 The Result of This Case Study

The results in Figure 3.4 and 3.5 shows that we can use an application such as the logging application in dom0 to capture information in memU or in the memory space of any process, or commands of appU such as *c1* in domU.

We now have an appropriate logging system to mitigate the risks associated with CSA threat 1. However, this system is in the IaaS which involves at least two parties including a provider and customer. Hence, the system's security needs to be analysed before deployment as described in Section 3.4.2.4.

3.4.2.4 Analysing the Security of our Logging System

The proposed logging system architecture in the implementation is systematically designed based on the logging components. Therefore, it inherits the advantages of the logging components. For example, one of the advantages is an architecture that is based on the logging components can be used as a tool to analyse the security of logging processes and log files, which are distributed across the IaaS visualisation environment. Thus, one can analyse the security of the logging system architecture in the implementation. We provide an approach to systematically analyse the security of logging systems before deploying them in the IaaS real world productions.

Ideally, these analyses needs to be completed before deploying our logging system. We can directly use the logging system architecture, Figure 3.3, as a tool for our analyses, as discussed in the lists below.

- **Analysis of the integrity issues of the logging system**

  From the architecture of the logging system in Figure 3.3, our system deploys P1 in the dom0 user level, this may lead to integrity issues of this logging system. This is because malicious insiders may modify P1's code to produce contents of log files which benefit themselves, as discussed in Section 3.3.3.1.

- **Analysis of the privacy issues with our logging system**

  The question is where F3 be located. From the architecture in Figure 3.3, it will be located in disk0. Again, malicious insiders may learn about, or alter F3, which is a privacy violation to a customer, as discussed in Section 3.3.3.2.

3.4.3 Possible Solutions to the Security Issue of F3

This section discusses possible solutions of the security issues of log files as F3 in the logging system in this case study.

Our logging system in the case study will be deployed in dom0, this promotes privacy concerns for domU as partially discussed in Section 3.4.2.4. Actually, P1 or libVMI can be placed in a VM such as a dom0 as in our case study, within a hypervisor, or within any other part of the virtualisation architecture as argued by [79].

Hence, the solution of the privacy issues above could be that a trusted third party or TTP, not a dom0 or provider, should handle our logging system. As a result, the TTP can manage and maintain the logging system (P1 or libVMI, the logging application or app0, and F3) in a special-privileged domU which operates this system. The security challenge of this new solution is how to maintain integrity of this TTP domU, for instance how to prevent the provider from altering or learning about F3.

The trusted computing and its related-research could be the solution. An example is the work of Rocha, Abreu, and Correia [52], which proposes a solution based on the TPM and offers protection against a malicious provider who has full privilege over a domU in an IaaS.

3.4.4 Ways To Detect Other Forms of Attacks of Threat 1

This subsection discusses how to detect other forms of attacks of threat 1 using logging solutions.

We use spammer concerns or spam activities as a representative of the forms of attacks enclosed in threat 1. To mitigate the risks associated with this threat, such as mis-use of domUs by criminals to facilitate those attacks, the idea is to have the ability to collect domU's activities from domUs to be used as evidence to identify the attacks. One approach to do so is to collect behaviour or activities of a process or processes in domUs. This is because the attacks eventually use a process or processes in the domU to conduct criminal activities. Thus, if we can collect appropriate and sufficient behaviour or activities of suspect processes as log files, these files could be used as evidence to assist in identifying these attacks.

Thus, it is possible to detect some other forms of attacks enclosed in threat 1, such as domUs that host downloads for illegal software. This can be achieved by detecting the process that downloads a certain unacceptable amount of incoming network package. It is possible to detect the downloading process's misbehaviours as we do with the spam process. Although more research is needed to deal with all possible forms of threat 1, this case study shows that it is possible to detect process's behaviours and then record them as log files to be used as evidence to identify malicious use of domUs.

3.5 Evaluation and Discussion of the Logging Components

This section provides an evaluation and discussion of the logging components. Subsection 3.5.1 provides an evaluation of the logging components against how these components satisfy their goals. Subsection 3.5.2 gives many other discussions of the logging components, which affect many aspects of logging systems in IaaS, as well as PaaS.

3.5.1 Evaluation of the Logging Components

The logging components' goals were already discussed in the beginning of Section 3.3. We evaluate the logging components against these goals.

To satisfy the first goal which is that the logging components can be used to instantiate new logging system architecture to mitigate the risks associated with all CSA threats in IaaS environment, we demonstrate how to instantiate a logging system to mitigate the risks associated with CSA threat 1 outlined in Section 3.4.2. This is an example of how the logging components can be used to instantiate logging systems that are capable of mitigating the risks associated with all CSA threats.

To satisfy the second goal which is that the security of the newly built logging system architecture, that is based on the logging components, can be systematically analysed before deployment, we discussed how to analyse the security of the newly designed logging system architecture in the same case study in Section 3.4.2.4. We then found out this logging system's privacy issues that need to be addressed before deployment.

We also consider the logging components as 'generic' because of these considerations. 1) One can choose the components especially Px and Fy from the logging components to instantiate their new logging systems' architecture. 2) Then, the security of the new logging systems can be analysed before deployment using the systems' architecture, from 1), as a tool.

3.5.2 Discussion of the Logging Components

Section 3.3.3 discusses how the logging components assist in analysing the security of logging systems. In this subsection, the discussions below provide many other analyses of the logging components, which affect many aspects of logging systems in IaaS, as well as PaaS.

The discussions include: any other locations of logging processes or Px and log files or Fy in the logging components; traditional log files versus Fy in the logging components; how do the logging components also facilitate the analysis of the security of Px; and applying the logging components to mitigate risks associated with CSA threats for PaaS.

The rest is discussion of: how the logging components facilitate faster logging system development with security concerns; flexibility and re-usability of development of Fy and Px; performance analysis of logging systems which are based on the logging components; and a TCB size measurement of a logging system in the cloud based on the logging components.

3.5.2.1 Any Other Locations of Logging Processes (Px) and Log Files (Fy) in the Logging Components?

We identify and describe all possible and important locations of Px and Fy of the logging components in the IaaS, as discussed in Section 3.3.2. However, a basic understanding of logging systems and the logging components in the Xen-based IaaS layers could apply to more complicated-based layers such as the layers discussed in [22; 46; 91]. These complicated-based layers can indicate new locations of Px and Fy apart from the locations which are presented in our logging components.

For example, [46] states that IaaS is composed of four layers: facility, network, hardware, and OS; thus it is possible to add Px and Fy into the network layer. However, we focus only on construction of the logging components based on a representation of only one machine or server, to have clear comprehension of the logging components inside a machine before it is connected to other machines to form the real world IaaS infrastructure. After the logging components for one machine is clearly understood, then we can consider to add a network layer to the logging components and may add Px or Fy into the network layer to form more complicated logging components with a network.

The construction of IaaS layers can vary in the literature. If we map the layers in the logging components (Figure 3.1) to [46]'s IaaS layers (facility, network, hardware, and OS), it can be seen that the 'hardware' layer in [46] is the 'hw0' layer in our logging components. The 'OS' layer in [46] is considered as a combination of the 'hypervisor, domU, and dom0' in our logging components. Therefore, in our logging components, there are only two layers which are the hardware and OS, compared to the four layers in [46]. It is possible to have Px and Fy in the facility layer. For example, if the facility is a keypad to access to a room that locates IaaS severs; thus it is possible to have Px and Fy in the keypad.

However, we focus on construction of the logging components based on the hardware and OS layer of the IaaS architecture, which are based on the layers discussed in [46]. We argue that our logging components can be expanded by adding any extra layer on top of or below the hardware and OS layer. As a result, one can identify locations of Px and Fy in the newly added layers such as the network layer. This is out of the scope of this thesis.

3.5.2.2 Traditional Log Files Versus Fy in the Logging Components

The logging components show the possible logical and physical locations of Fy. These locations can affect many security concerns of logging systems as discussed in Section 3.3.1 and 3.4.2.4. We did not propose a new approach to enhance the fundamental properties of log files such as non-repudiation or tamper evident. There is research such as [83] that outlines or proposes these properties which can be applied to implementation of Fy.

For example, to deal with the tamper evident property, Fy can be implemented by re-using existing logging file systems such as in the research done by Crosby and Wallach, which introduces the semantics of tamper-evident logs using a tree-based data structure [83]. Therefore, it could be possible to efficiently re-use existing Fy when building a new logging system, instead of building a whole new log file system. This should make a logging system reliable, its security analysable, and its development and deployment faster, more flexible, and rapidly adaptable in the IaaS real world.

3.5.2.3 How do the Logging Components also facilitate Analysis of the Security of Logging Processes (Px)

We have already discussed how the logging components facilitate the security analysis of Fy in Section 3.4.2.4. We also argue that the security of a logging process (such as P1 in the spamming case study in Figure 3.3) itself has to be systematically analysed before deployment. For example from Figure 3.3, the simple security relevant question is, how can customers ensure the integrity of P1 which is run by the provider in dom0 user level? Locating P1 in dom0 user level is a security risk because the providers or insiders may maliciously modify P1's code to produce contents of log files as F3, which benefit themselves.

The possible solutions: in the case study, the problem becomes more complicated because P1 deployed in dom0. This makes P1 is easier to be attacked by malicious insiders such as administrators who usually have full privilege over dom0 in IaaS environment. The possible solution is to have a proof to say that the P1 code is genuine. It may again trusted computing/TPM research which provides a function (called a remote attestation) to remotely verify the integrity of the computer platform to prove that some software such as P1 or hardware is genuine or correct. Work in [52] provides methods of the remote attestation in the IaaS cloud context. Therefore, we argue that logging system architectures that are based on the logging components can be used to highlight such security concerns of P1. These concerns have to be addressed before deploying P1 in the real world.

3.5.2.4 Performance Analysis of Logging Systems Which are Based on the Logging Components

This thesis presents only examples of how to analyse the security of a new logging architecture itself before deployment, in Section 3.4.2.4. However, in the complex environment of IaaS, performance of the new built logging system could also be a critical factor that needs to be thoroughly analysed before deployment. For example, from the logging components (Figure 3.1), the performance of a logging system that deploys P2 in dom0 kernel, and of one that deploys P4 in domU kernel, should be different and needs to be considered before deployment.

With knowledge of the locations of Px in the newly built logging system architecture, made clear by using the logging components, the system's performance analysis could be feasible. We believe that the logging components can be seen as a preliminary study to really achieve a complete analysis of the newly built logging systems, based on all possible aspects, such as security and performance.

3.5.2.5 Flexibility and Re-usability of Development of the Log Files (Fy) and the Logging Processes (Px)

To cope with the continuously changing, growing, or developing behavior of Internet-based services, [92] states that ideally effort and time commitments of the deployment of monitoring and evaluation software for Internet applications would be minimal. This perspective by [92] above should be applied to the development of logging systems in the cloud as well. Ideally, effort and time dedication of the development of logging systems should be as less as possible to cope with the dynamic behavior of the cloud. In addition, we agree that flexibility and re-usability of development of Fy and Px could be an important factor to minimise effort and time dedication of logging system development. This facilitates faster the logging system development, which will be discussed in Section 3.5.2.6.

For flexibility and re-usability, we argue that Fy can be implemented by re-using existing log files as partly discussed in Section 3.5.2.2. Fy can also be implemented and based on provenance concepts which become increasingly important as discussed in [28]. We have already demonstrated the re-using available logging processes (libVMI) as P1 can assist building the logging system in the case study, Section 3.4.2.1. This is possible because the logging components are implementation independent.

Moreover, the logging components present the logical and physical locations of Px and Fy in the IaaS structure. Then one can choose these components to build up their own logging system architecture. This flexibility and re-usability makes the development of logging systems faster, and more adjustable, as will be fully discussed in the next section (Section 3.5.2.6). Thus, we are always ready to deal with new emerging threats in the real world IaaS.

3.5.2.6 The Generic Logging Components Facilitate Faster Logging System Development With Security Concerns

There is some work focusing on the implementation of Fy such as [83] as discussed in Section 3.5.2.2. Moreover, other works also focus on implementation of Px such as all those that are discussed in the related work section (Section 2.8). This thesis does not focus on creating a new Px and/or Fy. We encourage the building of a logging system by following these two steps: 1) the design phase which can instantiate a logging system architecture by choosing the appropriate Px and Fy (from the logging components), which fit their needs; and 2) the implementation phase which can re-use the existing Px and/or Fy systems (if any) that fit the designed architecture in 1).

Thus, we argue that one can build their own logging systems based on the logging components with the security concerns of the systems themselves and their re-usability. However, it is also possible to create their own versions of logging systems without re-using, but the systems can still be based on the logging components as well. As a result, all participating parties can analyse how critical components (Px and Fy) of a newly built system are distributed in the IaaS visualisation environment, which affects the security of this system.

3.5.2.7 Applying the Logging Components to Mitigate the Risks Associated with CSA Threats for PaaS

Although this thesis focuses on dealing with the CSA threats to IaaS, these threats affect the security of PaaS and SaaS as well [23]. PaaS can always be built up by adding extra layers on top of IaaS layers as discussed in the beginning of Section 2.2. Therefore, comprehension of our proposed logging components and of logging systems in IaaS could also assist the mitigation of the risks associated with the CSA threats applicable to the security of PaaS.

For example, it is possible that ones can add the extra layers on top of our proposed logging components to build up the logging components for PaaS such as in [22] an integration and middleware layer can be added on top of IaaS to build up PaaS. Then, they can consider the possible and appropriate locations of Px and Fy in this new PaaS logging components. These new locations of Px and Fy in the PaaS architecture could affect their security concerns the same as we discussed for our logging components for IaaS in this thesis.

3.5.2.8 A TCB size measurement of a logging system in the cloud based on the logging components

[38] argues that the TCB size of a software system (N.B. a logging system is also a

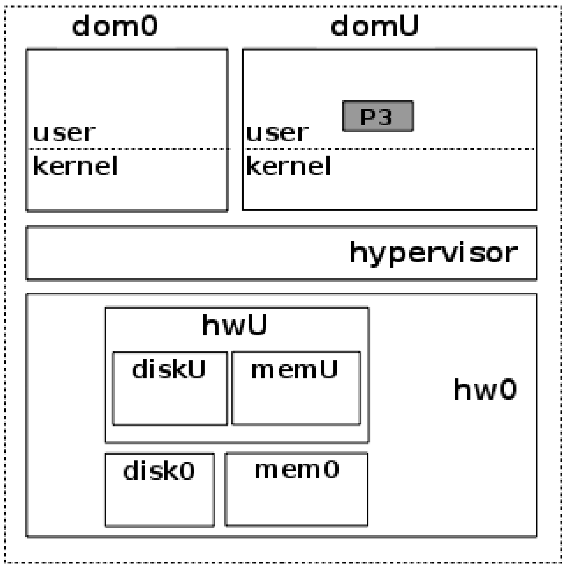


Figure 3.6: The biggest TCB size, the dot rectangle

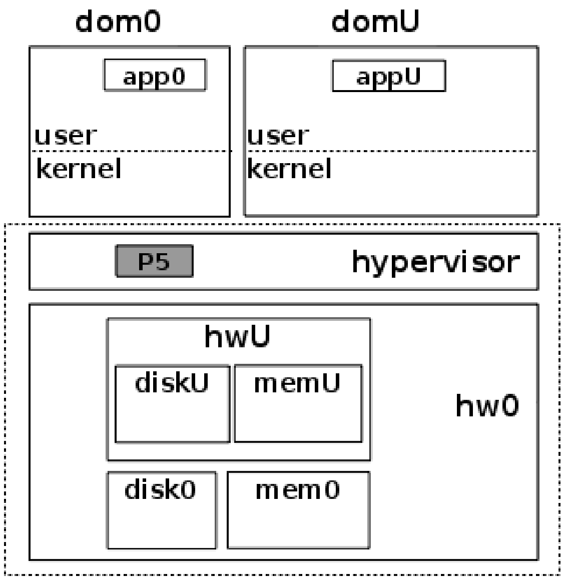


Figure 3.7: The smallest TCB size, the dot rectangle

software system) can be used to evaluate the trustworthiness of that system. We can define the size of TCB of a logging system based on the logging components as the set of all hw, a hypervisor, dom0, and domU. TCB size of a logging system can be compared to the TCB size of another logging system. For example, the TCB size of a logging system, which includes all components (hw, a hypervisor, dom0, and domU) is bigger than the TCB size of another logging system, which includes only hw, a hypervisor, and dom0.

The biggest TCB size of a logging system is when this system deploys P3 in domU user level (Figure 3.6). In this case, this TCB size includes hw0, hypervisor, dom0, and domU. The size includes domU because this deploys P3. Figure 3.7 is the smallest TCB size of a logging system, when the system deploys P5 in hypervisor. In this case this TCB size includes only hw0 and hypervisor. The size does not include dom0 and domU because there is no any logging process is deployed in dom0 and domU. Thus, if all logging processes are only in the hypervisor, the TCB size includes only the hw0 and hypervisor. This TCB size is the smallest one.

3.6 Conclusions

This chapter addressed Gap 1 (c) which is the lack of security analysis of the logging systems themselves before the deployment of the systems in to the IaaS real world productions, as discussed in Section 1.2. The previous chapter provided a holistic representation of the environment of logging systems in the cloud. This representation is utilised as the basis for the design of a generic framework of logging solutions to mitigate the risks associated with CSA threats (Objective 2). We call this framework **generic logging components for infrastructure as a service (IaaS) cloud**.

Thus, the main contribution of this chapter was these generic logging components. To facilitate the systematic support for accountability in the cloud, these generic logging components provide ways to build logging systems. The value of the generic logging components is to encompass all possible instantiations of logging solutions for the IaaS cloud, and to provide a clear view of all components that relate to logging systems in IaaS.

This view provides a basis for the analysis of logging systems' security before their deployment. Thus, the generic logging components enable logging systems to be appropriately designed or manipulated by participating cloud parties including a provider, customer, or auditor. As as result, this enhances systematic support for accountability in the could.

In the next chapter (Chapter 4), the generic logging components are utilised as the basis for describing the related work in the form of patterns. These patterns can be made up of the logging components or can be associated to the logging component configurations. The advantages and disadvantages encountered when using different patterns can be used to clarify the fact that a number of patterns and logging system architectures based on these patterns are missing, for example, our proposed logging system in the spamming case study in Section 3.4.2. The proposed system is based on a pattern that is quit specifically more easily to be deployed, but it is not very robust. Chapter 4 also discusses a "spectrum" of patterns for describing how to construct logging systems of varying quality. It also presents sophisticated examples or case studies to illustrate and evaluate the proposed patterns.

The generic logging components are also utilised (in Chapter 5) as the basis for an analysis of how real world threats, specifically CSA threat 1, affect both the customer and provider simultaneously. It is also used as the basis for design of the proposed logging solutions in mitigating the risks associated with threat 1, in order to benefit both the customer and provider sides.

Chapter 4

Logging System Architectures and Patterns

This chapter mainly addresses research Gap 4, which is the lack of descriptions of logging systems in terms of design patterns of the systems' components. Its main contribution is three proposed design patterns in the context of logging in IaaS cloud. The proposed patterns facilitate analysis of logging systems.

The proposed patterns can also bring a number of benefits as the same as benefits from design patterns in object-oriented software design and development area. Thus, the proposed patterns could increase reusability of the design and development of logging systems. Moreover, designers should access the proposed patterns more easily. Additionally, the proposed patterns could assist a designer adopts design approaches which make a logging system reusable and not to choose approaches which do not concern reusability concepts. Lastly, they proposed patterns can also enhance the documentation and maintenance of existing logging systems.

We provide a spectrum of patterns for describing how to construct logging systems with varying characteristics. For developers, when building a logging system, the knowledge of characteristics of this system could assist them to get the right design of the system with minimal effort and time commitments. We also clarify why a number of patterns and logging system architectures based on these patterns are missing. To the best of our knowledge, these three logging patterns are not yet described in the literature.

Thus, this chapter intends to define, identify, and draw conclusions on the advantages and disadvantages of logging system patterns. Then it analyses existing works in relation to the patterns. These patterns can be made up of the generic logging components in Figure 3.1 or can be associated to the logging component configurations. This chapter also discusses a "spectrum" of patterns for describing how to construct logging systems of varying quality or characteristics. It also presents sophisticated examples or case studies to illustrate and evaluate the proposed patterns.

Section 4.1 investigates and introduces 93 possible architectures of a logging system in an IaaS. These architectures are divided into three categories. It also discusses the structure and example of architectures of each category in details. Associated with the possible architectures, Section 4.2 identifies and gives conclusions on the advantages and disadvantages of logging system patterns. First of all, the section discusses a definition of a pattern in general and in object-oriented software design and defines a pattern in logging system design and development. It then identifies and discusses three patterns for logging systems. After that, it gives conclusions on the advantages and disadvantages of the patterns.

Then, Section 4.3 analyses existing logging and monitoring works and our proposed system in relation to the patterns. To do so, it discusses advantages and disadvantages or characteristics of five existing systems, as well as our proposed system for spam. Lastly, the chapter is briefly summarised and concluded in Section 4.4.

4.1 All possible architectures of a logging system

This section investigates and introduces 93 possible architectures of a logging system in an IaaS. These architectures are divided into three categories. It also discusses the structure and example of architectures of each category in details.

All these possible architectures are formed based on the critical components or Px and Fy in the generic logging components in Figure 3.1. These components are in three domains: dom0, domU, and hypervisor. P1 and P2 are in dom0 user level and kernel level respectively. P3 and P4 are in domU user level and kernel level, respectively. Lastly, P5 is inside a hypervisor. For Fy, it is assumed that if a logging system architecture deploys F1 in diskU or F3 in disk0, it then needs to eventually deploy F2 in memU or F4 in mem0 respectively. The deployment of Fy is already discussed in Section 3.3.1.

Any concrete logging system will be based on one of the 93 possible architectures discussed in Section 4.1.1 to 4.1.3. Based on our investigation in the literature in this thesis, a few of the possible architectures exist. However, the non-existing ones can be interesting. For example, an architecture that deploys all nine Px and Fy, it is the most complicated architecture that is not existing and will be discussed in Section 4.1.3. One of the architecture's advantages is having many abilities to facilitate logging tasks including: to record the necessary logging data as log files across domU and dom0; and/or to assist in collaboration of Px and Fy with other components such as database components.

Reducing TCB size of a logging system is one of the three aspects of systematic support for accountability in IaaS, as discussed in Section 1.1, 2.13, and 3.5.2.8. One of disadvantages of the most complicated architecture is having a big TCB size of the system. This is because Px and Fy are deployed and distributed across all the three domains (domU, dom0, and hypervisor).

In contrast, for the most simple architecture as will discussed in Section 4.1.1, the architecture may not be able to record the necessary logging data across domU and dom0 compared to the most complicated one. The system TCB size is smaller than the TCB size of the most complicated one.

To simplify the presentation of all the possible architectures, they are divided into three categories called: single domain, two domains, and three domains. A single domain category means that all Px of a logging system are deployed in either dom0, domU, or a hypervisor. A two domains category means that all Px of a logging system are deployed in two domains among dom0, domU, or a hypervisor. A three domains category means that Px are deployed in all three domains, thus at least one Px of a logging system is deployed in each domain.

The 93 possible architectures compose of: 21 architectures in the single domain category, 45 architectures in the two domains category, and 27 architectures in the three domains category. The following subsections will clarify how each category gets the number of architectures. Below are conditions of all categories:

1. Each logging system architecture from these three categories has to deploy Px based on its category's conditions which will be discussed in the following sections. Eventually, the system has to deploy Fy by choosing one from these three different approaches of deploying Fy: in disk0, in diskU, or in both disk0 and diskU which we call disk0U.

2. When a system is deployed in dom0, its Px can be in: dom0 user level as P1, dom0 kernel level as P2, or both of them

3. When a system is deployed in domU, its Px can be in: domU user level as P3, domU kernel level as P4, or both of them.

4. When a system is deployed in a hypervisor, its Px can be in only this hypervisor as P5.

5. The notation PaPb such as P1P2 means that when an architecture or a system deploys both Pa and Pb. Thus, P1P2 means that when an architecture or a system deploys both P1 and P2.

6. PbPa has the same meaning as the meaning of PaPb.

7. In forms such as Pa/Pb, Pa/disk0, PaPb/PcPd, or PaPb/Pc/Pd/diskU, '/' is a separator notation among the elements of the forms. For example, Pa/disk0 indicates that Pa is in a domain and disk0 is deployed by a system that deploys Pa.

4.1.1 The single domain category

This section discusses the structure and example of architectures of this category in details. We form an architecture of this category by firstly considering the deployment of Px of a system in either dom0, domU, or a hypervisor, as discussed in the conditions above. After that, to create a final architecture of this system, the system can choose to deploy Fy in appropriate locations: diskU, disk0, or both of them.

For considering of deployment of Px, when Px is deployed in a domain, this creates one or more forms of deployment of Px. For example, when a system deploys one Px (such as Pa) in a domain, this creates one form of deployment as: a system which deploys Pa called a Pa form. When a system deploys two Px (such as Pa and Pb) in a domain, this creates three forms of deployment as: a system which deploys Pa called a Pa form; a system which deploys Pb called a Pb form; and a system which deploys both Pa and Pb (PaPb) called a PaPb form.

After that, to create a final architecture of a system, each form above can choose to deploy Fy in disk0, diskU, or disk0U. For example, a Pa form can deploy disk0 to create a final architecture of a system which deploys Pa and disk0. We represent this final architecture as 'Pa/disk0'. Thus, each form can create three final architectures. For example, a Pa form creates three final architectures as: Pa/diskU, Pa/disk0, and Pa/disk0U.

Figure 4.1 presents all 21 possible final architectures of the single domain category. From the figure, each branch can be a final architecture of a logging system, for example, see the three shaded boxes with the dotted-lines labelled one. The lines create the branch of dom0, P1, and disk0. This branch is a final architecture of a logging system, and this architecture is represented as dom0/P1/disk0 or for short P1/disk0. The representation means that a logging system that follows this architecture deploys P1 in dom0 user level, F3 in disk0, and F4 in mem0. The architecture is presented in Figure 4.2.

Forming all the 21 final architectures or branches is discussed in the lists below:

- 9 architectures when a system deploys Px in dom0

  See the three branches originated from dom0 box in Figure 4.1, this creates three forms as: a system which deploys P1, a system which deploys P2, and a system which deploys P1P2. These three forms can deploy the three deployment approaches of Fy or condition 1 discussed above. This then creates nine architectures as: P1/disk0 (in the figure from dom0 box, see P1 box and its first branch), P1/diskU, P1/disk0U, P2/disk0, P2/diskU, P2/disk0U, P1P2/disk0, P1P2/ diskU, and P1P2/ disk0U

- 9 architectures when a system deploys Px in domU

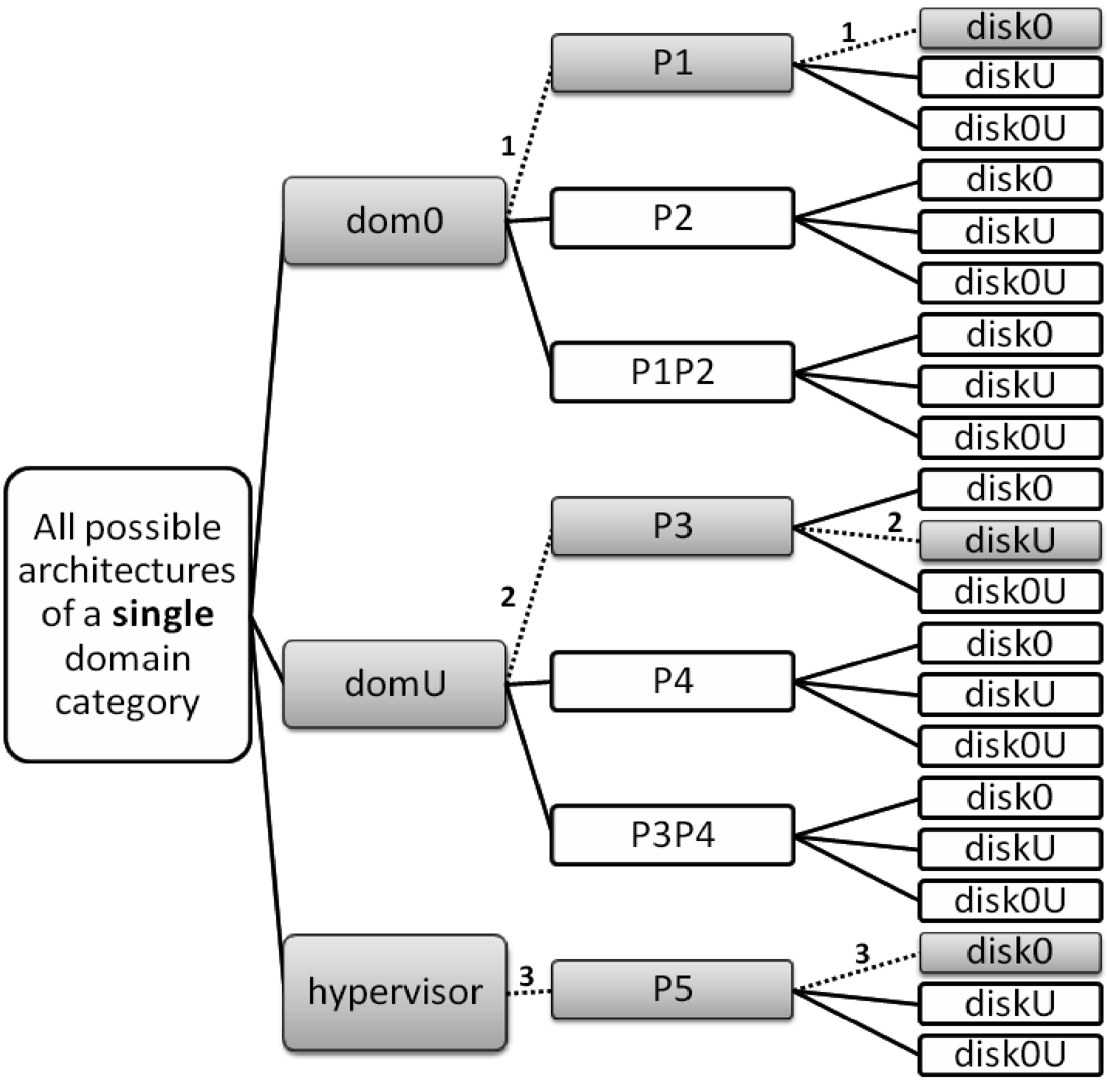


Figure 4.1: All possible architectures of a single domain category

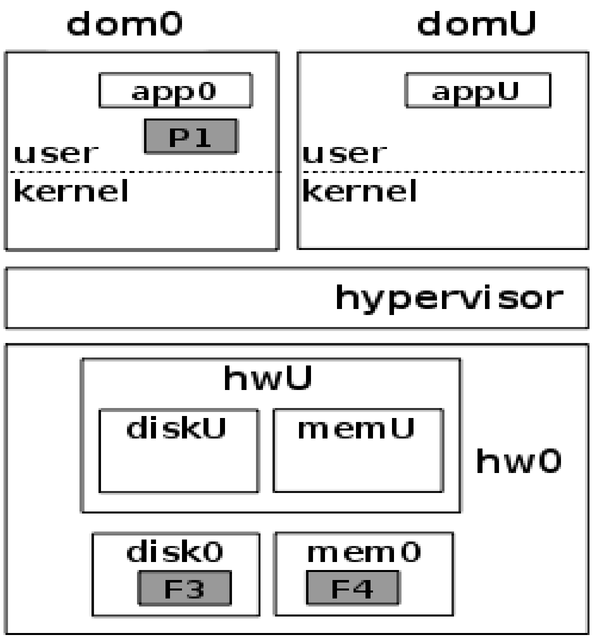


Figure 4.2: A logging system architecture deploying P1, F3, and F4 (dom0/P1/disk0 or for short P1/disk0)

See the three branches originated from domU box in Figure 4.1, this creates other three forms as: a system which deploys P3, a system which deploys P4, and a system which deploys P3P4. These three forms can deploy the three deployment approaches of Fy. This then creates other nine architectures as: P3/disk0 (in the figure from domU box, see P3 box and its first branch), P3/diskU, P3/disk0U, P4/disk0, P4/diskU, P4/disk0U, P3P4/disk0, P3P4/diskU, and P3P4/ disk0U

- 3 architectures when a system deploys Px in a hypervisor

  Lastly, see the branch originated from hypervisor box in Figure 4.1, this creates only one form which is a system which deploys P5. The form can deploy the three deployment approaches of Fy. Then, this creates other three architectures as: P5/disk0, P5/diskU, and P5/disk0U.

Not all 21 architectures of the single domain category already exist. We find three existing architectures, which are listed below. Some of the architectures are from the related work concerning logging systems in IaaS in Section 2.8.

1. The architecture in Figure 4.2 is the architecture of the logging system (Figure 3.3) in the spamming case study in Section 3.4.2.1.

2. The dotted-lines labelled two in Figure 4.1 create the branch of domU, P3, and diskU. This is the JAR logging [76] system architecture.

3. The dotted-lines labelled three in Figure 4.1 create the branch of hypervisor, then P5, and disk0. This architecture is for AVMs [44] system and a system in [93].

Systems of the other 18 non-existing architectures are feasible to be built. We do not investigate if all these systems are very useful. However, these architectures can be used as tools for analysis of systems (e.g., in terms of systems' robustness) that built based on these architectures. For example, from Figure 4.1, the following non-existing and existing architectures below can be analysed and compared.

When following the branch dom0/P2/disk0 in Figure 4.1, this is the first architecture or P2/disk0 non-existing architecture. When following the branch dom0/P1/disk0, the second one is existing P1/disk0 architecture. A system built based on the architecture of P2/disk0 should be more robust compared to the architecture of P1/disk0. Then, for example, it will be more difficult for attackers to compromise P2 in the kernel of the first architecture, compared to compromising P1 in the user level of the second architecture. Full analysis of systems that built based on these non-existing architectures is out of the scope of this thesis.

4.1.2 The two domains category

This section discusses the structure and example of architectures of this category in details. A two domains category means that all Px of a logging system are deployed in two domains among the three domains (dom0, domU, or a hypervisor). Figure 4.3 presents the 45 architectures of this category. For example, the dotted-lines labelled one in the figure create the branch of dom0U, P2, P4, and disk0U. This branch represents a logging system architecture as P2/P4/disk0U.

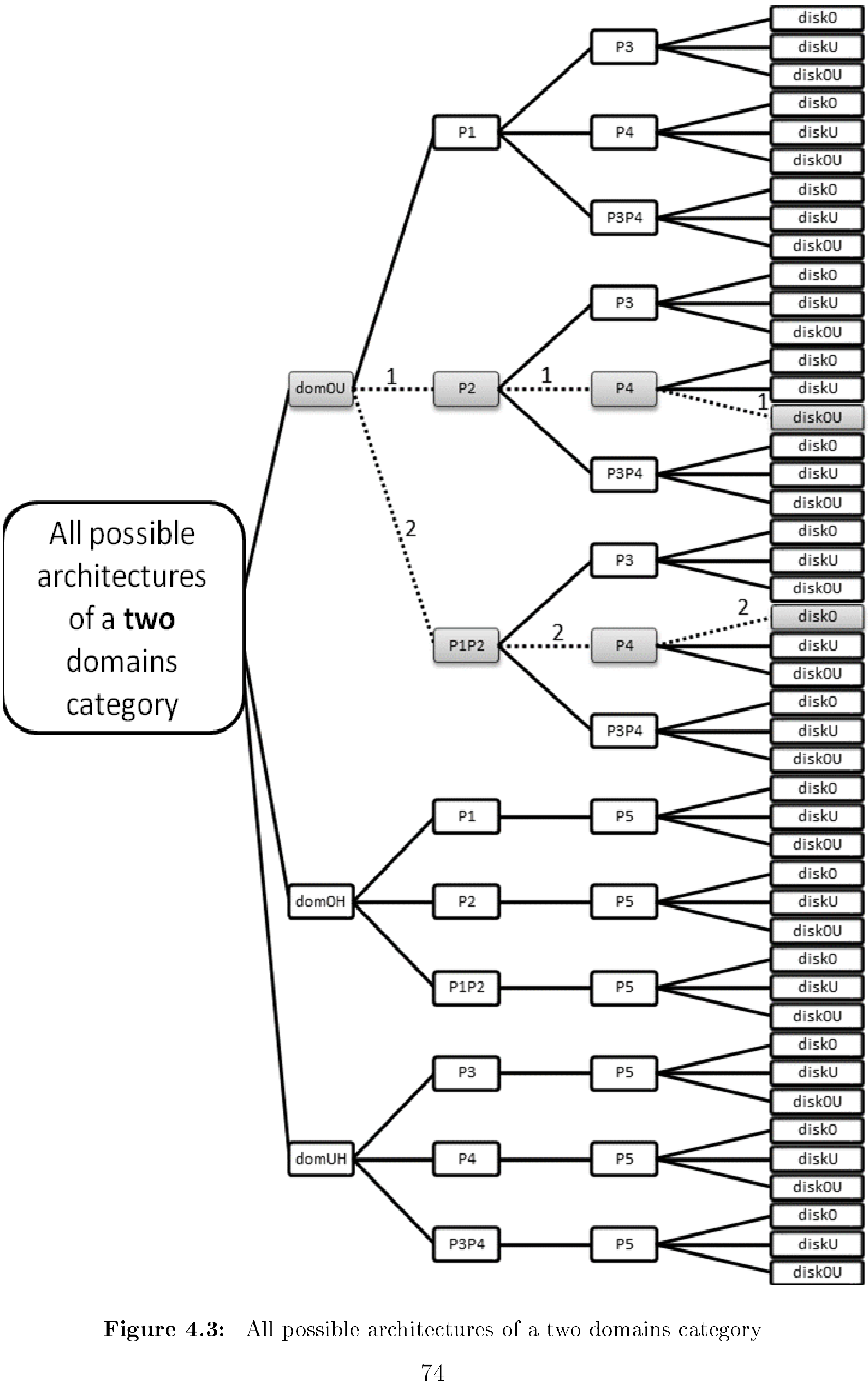


Figure 4.3: All possible architectures of a two domains category

This means that the architecture deploys P2 in dom0 kernel, P4 in domU kernel, F1 in diskU, F2 in memU, F3 in disk0, and F4 in mem0. The architecture is presented in Figure 4.4.

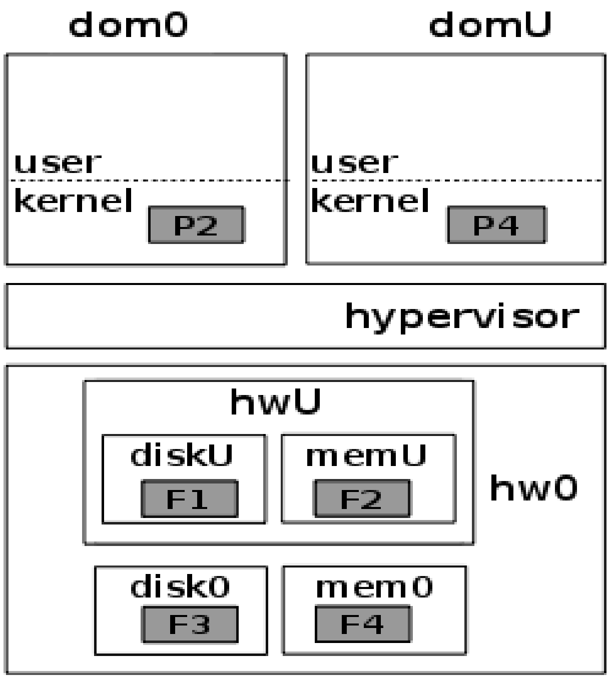


Figure 4.4: A logging system architecture deploying P2, P4, and F1 to F4

A box which labelled disk0, diskU, or disk0U is the end of the representation of a final architecture. For example, from the first stack of boxes from the right of Figure 4.3, the top box of the stack is disk0 box which can be repetitively linked back to P3, P1, and dom0U box. This representation is a dom0U/P1/P3/disk0 or P3/P1/disk0 architecture and applies to the discussions of the next category.

We form an architecture of this category by firstly considering the deployment of Px of a system in the first domain then in the second domain. Finally, to create a final architecture of this system, the system can choose to deploy Fy in disk0, diskU, or disk0U, see condition 1. This section represents a Pa form as just Pa, Pb form as Pb, and PaPb form as PaPb. For example, a P1 form, P2 form, and P1P2 form are represented as P1, P2, and P1P2, respectively. Dom0U is an abbreviation which means that Px are deployed in both dom0 and domU.

Dom0H and domUH are also abbreviations which mean that Px are deployed in both dom0 and hypervisor (H) and in both domU and hypervisor, respectively.

Below are discussions of forming the 45 architectures.

- 27 architectures when deploying Px in dom0 then in domU (dom0U)

  There are three forms of deploying Px in dom0: P1, P2, and P1P2 see the three branches originated from dom0U box in Figure 4.3. Then, each form can deploy other three forms of domU: P3, P4, and P3P4 see the three branches originated from each of boxes labelled P1, P2, or P1P2 after box dom0U in the figure. Thus, this generates nine forms of dom0U: P1/P3, P1/P4, P1/P3P4, P2/P3, P2/P4, P2/P3P4, P1P2/P3, P1P2/P4, and P1P2/P3P4.

  Finally, these nine forms can choose to deploy Fy by the three different approaches or condition 1. When multiplying these 9 forms by the 3 different approaches of deploying Fy, this generates the 27 final architectures. These architectures are in Figure 4.3, from the first stack of boxes from the right of the figure see the first 27 boxes from the top of the stack.

- 9 architectures when deploying Px in dom0 then in hypervisor (dom0H)

  There are three forms of deploying Px in dom0: P1, P2, and P1P2 see the three branches originated from dom0H box in Figure 4.3. Then, each form can deploy a form of hypervisor as P5 see the branch originated from each of boxes labelled P1, P2, or P1P2 after box dom0H in the figure. Thus, this generates 3 forms of dom0H: P1/P5, P2/P5, and P1P2/P5.

  These three forms can choose to deploy Fy by the three different approaches. When multiplying these 3 forms by the 3 different approaches of deploying Fy, this generates the 9 final architectures. They are in Figure 4.3, from the first stack of boxes from the right of the figure see the 10th to 18th boxes from the bottom of the stack.

- 9 architectures when deploying Px in domU then in hypervisor (domUH)

  There are three forms of deploying Px in domU: P3, P4, and P3P4 see the three branches originated from domUH box in Figure 4.3. Then, each form can deploy a form of hypervisor as P5 see the branch originated from each of boxes labelled P3, P4, or P3P4 after box domUH in the figure. Thus, this generates three forms of domUH: P3/P5, P4/P5, and P3P4/P5.

These three forms can choose to deploy Fy from the three different approaches. This also generates 9 final architectures. They are in Figure 4.3, from the first stack of boxes from the right of the figure see the first nine boxes from the bottom of the stack.

We find two existing architectures of this category which are listed below.

1. The architecture in Figure 4.4 is the architecture of Flogger [30] which is the branch of P2/P4/disk0U.

2. The dotted-lines labelled two in Figure 4.3 create the branch of P1P2/P4/disk0. This is PASSXen [43] system architecture which is Figure 4.5.

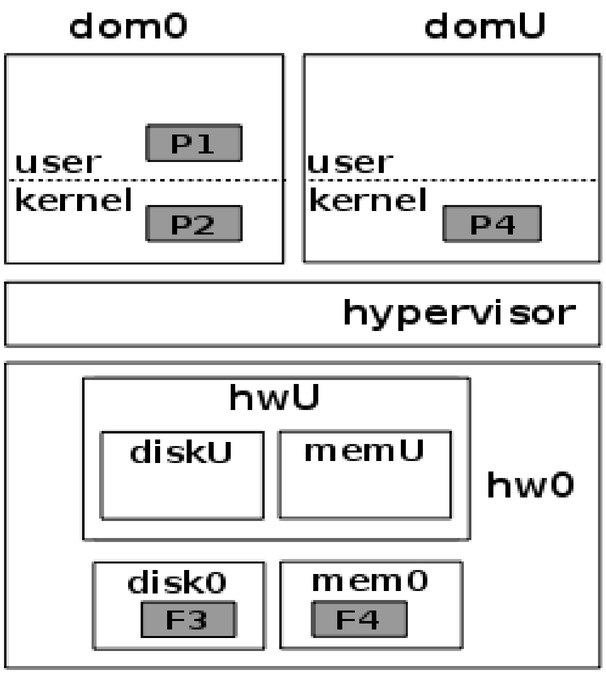


Figure 4.5: An architecture of PASSXen

Systems of the other 43 non-existing architectures are feasible to be built. We do not investigate if all these systems are very useful. However, these architectures can be used as tools for analysis of the systems (such as in terms of robustness of the systems) that built based on them. For example, from Figure 4.3, the two following non-existing architectures can be analysed and compared to.

When following the branch dom0H/P2/P5/disk0 in Figure 4.3, this is the first architecture which is P2/P5/disk0. When following the branch dom0U/P1/P3/disk0 in the figure, the second one is P1/P3/disk0 architecture. A system built based on the first architecture should be more robust compared to a system built based on the second architecture. Then, for example, it will be more difficult for attackers to compromise P2 in the kernel and P5 in hypervisor of the first architecture, compared to compromising P1 and P3 in the user levels of the second one. Full analysis of systems that built based on these non-existing architectures is out of the scope of this thesis.

4.1.3 The three domains category

This section discusses the structure and example of architectures of this category in details. A three domains category means that Px are deployed in all three domains or that at least one Px of a logging system is deployed in each of the three domains. Figure 4.6 presents 27 possible architectures of the catalogue.

We form an architecture of this category by firstly considering the deployment of Px of a system in the first, second, then third domains. Finally, to create a final architecture of this system, the system can choose to deploy Fy with the three different approaches or condition 1. Dom0UH (the second box from the left of Figure 4.6) is an abbreviation which means that Px are deployed in dom0, domU, and hypervisor.

For example, the dotted-lines labelled one in the figure create the branch of P1/P3/P5/disk0. It is a logging system architecture which deploys P1 in dom0 user level, P3 in domU user level, P5 in a hypervisor, F3 in disk0, and F4 in mem0. The most complicated architecture can be the one when following the dotted-lines labelled two in Figure 4.6. This is P1P2/P3P4/P5/disk0U architecture which deploys all the nine critical logging components or the five Px and four Fy.

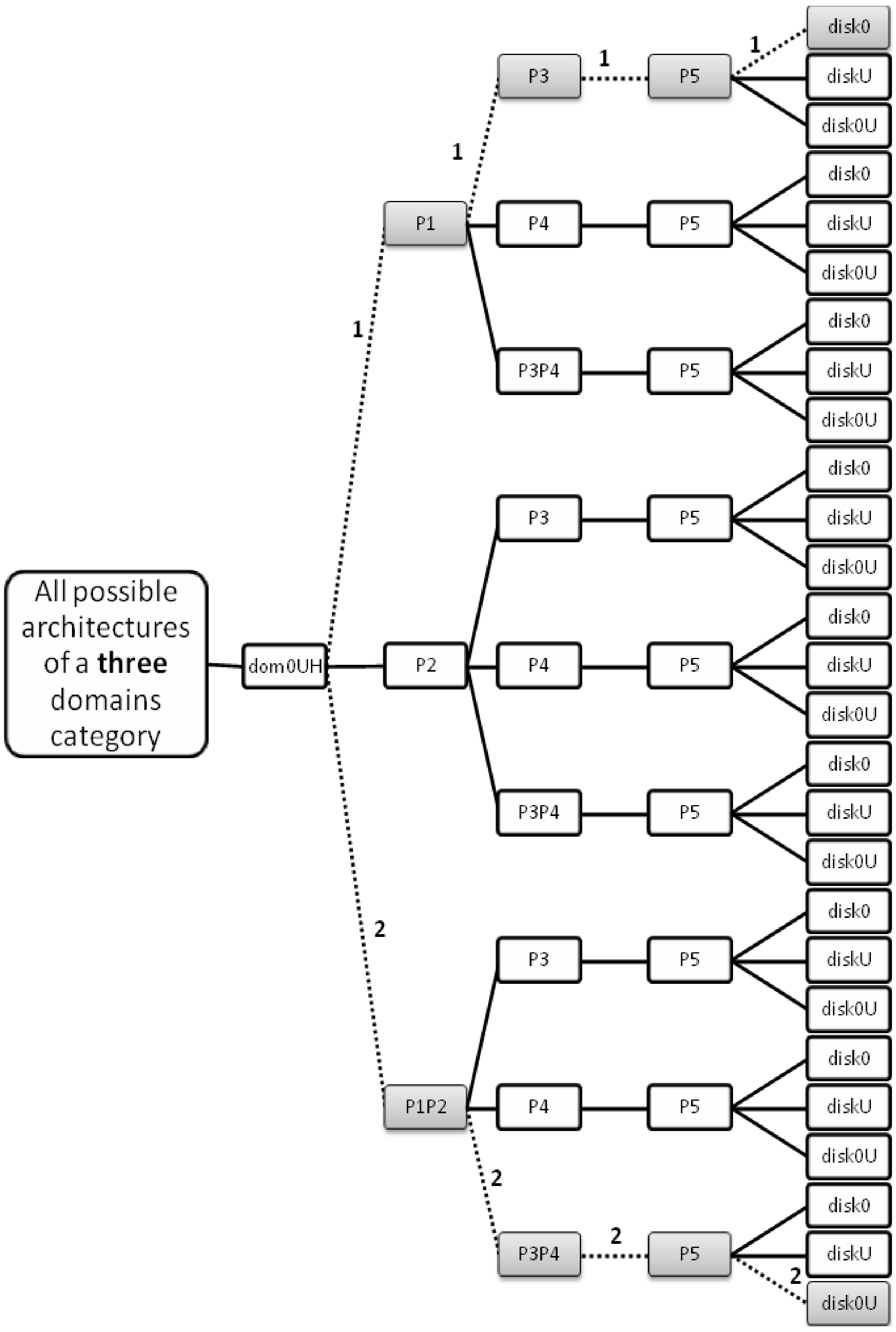


Figure 4.6: All possible architectures of a three domains category

To form all the 27 architectures, there are three forms of deploying Px in dom0: P1, P2, and P1P2 see the three branches originated from dom0UH box in Figure 4.6. Then, each form can deploy other three forms of domU: P3, P4, and P3P4 see the three branches originated from each of boxes labelled P1, P2, or P1P2 after box dom0UH in the figure. Thus, this generates nine forms: P1/P3, P1/P4, P1/P3P4, P2/P3, P2/P4, P2/P3P4, P1P2/P3, P1P2/P4, and P1P2/P3P4.

After that, each form can deploy a form of domH as P5 see the branch originated from each of boxes labelled P3, P4, or P3P4 after the boxed labelled P1, P2, or P1P2 in the figure. Thus, this generates nine forms (see the nine boxes labelled P5): P1/P3/P5, P1/P4/P5, P1/P3P4/P5, P2/P3/P5, P2/P4/P5, P2/P3P4/P5, P1P2/P3/P5, P1P2/P4/P5, and P1P2/P3P4/P5.

Finally, these nine generated forms can choose to deploy Fy with the three different approaches. When multiplying these 9 forms by the 3 different approaches of deploying of Fy, this generates the 27 final architectures. These architectures are in Figure 4.6, see all the boxes of the first stack of boxes from the right of the figure.

We did not find existing architectures of this category yet. Systems of these architectures are feasible to be built. We do not investigate if all these systems are very useful. However, as the same as the non-existing architectures of the first two categories, the architectures in this category can be used as tools for analysis of systems that built based on them.

4.2 Logging System Patterns in IaaS

This section defines and identifies logging system patterns. It then gives conclusions on the advantages and disadvantages of the patterns. It starts by discussions of a definition of a pattern in general and in object-oriented software design. It then defines a pattern in logging system design and development area. The section then identifies and discusses three patterns for logging systems. Finally, it gives conclusions on the advantages and disadvantages of the patterns.

4.2.1 Defining Logging System Patterns

This section gives a definition of a pattern in general and in object-oriented software design. Then, it defines a pattern in logging system design and development in IaaS.

4.2.1.1 Definition of a pattern in general and in object-oriented software design

In general, [49] describes a design pattern as "a documented best practice or core of a solution that has been applied successfully in multiple environments to solve a problem that recurs in a specific set of situations." Based on simple and elegant solutions to specific problems in object-oriented software design, [50] design patterns (also called by [94] as software design patterns) are defined as:

"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

[49] states that patterns typically comprise of a name, a purpose, a description of when and why to apply the pattern, structural diagrams, examples of use, and a discussion of interactions with other patterns.

Gamma et al [50] discuss a number of benefits of the design patterns. They state that: "design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design right faster".

The design patterns in logging context could bring the same benefits as above. These benefits could be: to promote the reusability of design and development of logging systems; to make the logging design patterns more accessible to developers of new logging systems; to assist a designer chooses design alternatives that make a logging system reusable and avoid alternatives that compromise reusability; and to improve the documentation and maintenance of existing logging systems by providing an explicit specification of the critical logging components (Px and Fy) with their locations in IaaS infrastructure and the components' underlying intent. To sum up, logging patterns could assist a designer gains a design right faster.

Apart from the object-oriented area, design patterns are also applied in other environments [49]. For example, with only trivial changes, a design pattern description can be adapted to refer to software design patterns in general [49]. Thus, design patterns should be applicable in the area of the logging system design and development in IaaS as well. We define and identify patterns for this area.

4.2.1.2 Defining a pattern in logging system design and development

For simplicity, we call a design pattern of logging system design and development in IaaS as a 'pattern'. We will define its definition from design patterns discussed by Gamma et al in Section 4.2.1.1. Their definition of a design pattern is:

descriptions of ***communicating objects and classes*** *that* ***are customized*** *to solve a general design problem in a particular context.*

For further discussions and based on the generic logging components, our general definition of a design pattern or pattern in logging system design and development environment is:

descriptions of ***participating critical components (Px and Fy) and their locations*** *that* ***are customized (the components can be appropriately located in IaaS components including domU, dom0, and hypervisor)*** *to solve a general design problem in a particular context.*

Participating critical components are Px and Fy that are used to form a logging system.

From the generic logging components in Figure 3.1, the logging processes or

P1 to P5 and log files or F1 to F4 are critical components of a logging system, as discussed in Section 2.9 and 3.2. Our patterns can be used to describe or compare between logging systems in IaaS in terms of their characteristics or advantages and disadvantages. This should facilitate design and development of the systems.

Figure 3.3 is an architecture of a logging system. Based on our investigation, this architecture is only one of all the 93 possible logging system architectures which are discussed in Section 4.1. A software design pattern can be used as a blueprint to create a concrete software architecture before building the software. In the logging system context, a concrete logging system architecture can be derived from a pattern as well. An absolute number of locations of Px and Fy deployed in a concrete system architecture derived from any of our patterns depends on the requirements and the context of the system. We identify patterns in IaaS logging environment in Section 4.2.2.

4.2.2 Identifying patterns for logging system design and development

This section identifies and discusses our three patterns for logging system design and development in IaaS.

Heer and Agrawala [94] identify software design patterns for information visualization based upon a review of existing frameworks and their own experiences building visualization software for the domain of information visualization. This section identifies patterns in IaaS logging environment based on the idea that we need a logging system that is simple and cheap for design and development, can facilitate enforcement of security policy of a logging system itself, or is capable of capturing as much as the necessary logging data as possible.

To achieve this idea, we follow three approaches. Based on the generic logging components of Figure 3.1, the first one is to investigate and evaluate all possible forms of distribution of Px and Fy (to form a logging system) into a customer side structure and a provider side structure. The next one is to investigate and evaluate the distributions of Px and Fy in system architectures of the related work concerning logging systems in IaaS in Section 2.8. The last one is to leverage our experiences building the prototype of logging systems in the spamming case study in Section 3.4.2.

The following sections describe our three identified patterns called: simple logging system, security facilitator logging system, and complex logging system. In the object oriented software development area, [50] describe their well-known design patterns using a consistent format. We follow this format to describe our patterns. The elements of the format are: pattern name, intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, known uses, and related patterns.

The common abilities of a logging system based on any of the three patterns are to capture and store necessary logging data. We define the meaning of this necessary logging data as: behaviour or activities of a process or processes in domU or as: a domU file's life cycle. An example of the data of behaviour or activities of a process or processes is discussed in Section 3.4.2 (the spamming case study) and Section 3.4.4. The example of data of a domU file's life cycle can be the creation, access, and destruction of the file, as discussed in [43]; or tracing domUs' data and files since they were created until deletion, as discussed in [27]. Chapter 5 fully discusses this data.

Note that, an absolute number of locations of Px and Fy deployed in a concrete system architecture derived from any of our patterns depends on the requirements and the context of the system.

4.2.2.1 Simple Logging System Pattern

The elements below are the descriptions of the simple logging system pattern.

1. **Pattern Name:**
   Simple Logging System

2. **Intent**
   Provide a simple logging system architecture for capturing the necessary logging data then storing the captured data as log files, which can decrease effort and time of design and development of the system.

3. **Motivation**

- **Problem** The changing, growing, or developing behaviour of Internet-based services are continuous as argued by Parkin and Morgan [92].

  They also state that ideally effort and time commitments of the deployment of monitoring and evaluation software for Internet applications would be minimal. Minimizing the effort and time commitments benefit rapid logging system development which is an important aspect of the development of logging systems in the cloud, as discussed in Section 3.5.2.6

- **Solution** The solution is to minimize effort and time commitments of design and development of logging systems. This can be achieved when the system deploys Px and Fy in either only the provider side or only the customer side.

  This does not include the deployment of Px as P5 in a hypervisor. To do so needs to modify the hypervisor. The deployment is complicated. The work of [93] apply this deployment. However, the output of their system is only small parts of the necessary logging data. The output is not sufficient to be considered as the necessary logging data. PASSXen [43] also do not publicly publish the code of their work, which is necessary to apply this deployment.

  Moreover, the source code of the present hypervisors including VMware and Xen are large and complex [95]. Then, to modify a hypervisor for deployment of Px in it is a challenge. The challenge can increase effort and time commitments of the deployment of a logging system. The deployment is opposite to the aim of this pattern. The aim of this pattern is to minimize effort and time commitments of design and development of logging systems. Therefore, the pattern does not include the deployment of Px as P5 in a hypervisor.

- **How and why the pattern works, and its example scenario**

  When Px and Fy of a system are in the same side, this is easier for design and development of the system compared to when the components are distributed across both sides.

  In the context of identifying spam activities in an IaaS, an example scenario of a customized concrete system for the context is the logging system in Figure 3.3, discussed in Section 3.4.2. This system records spam activities in domU, and is a simple logging system architecture derived from this pattern. P1 is installed in dom0 user level to capture the necessary logging data from domU. The captured data is stored as log files F3. Px (P1) and Fy (F3) are both on the provider side. This minimizes effort and time commitments of design and deployment of the logging system.

4. **Applicability**

   Use the pattern when:

   - To cope with rapid growing of an IaaS, the logging system development also need to be fast.
   - Logging systems are located in an IaaS visualisation infrastructure which involves more than one party. The infrastructure is normally complicated. To place Px and Fy diffusely in many locations in the infrastructure may cause difficulties in terms of design and development of the systems.
   - One wants quick, cheap, and simple ways to design and develop logging systems in an IaaS.

5. **Participants**

   - Provider side deployment
     - P1, P2: processes that individually or co-operationally capture the necessary logging data and store the captured data as log files.
     - F3: log files to be used to store the data captured by P1 and/or P2.
   - Customer side deployment
     - P3, P4: processes that individually or co-operationally capture the necessary logging data and store the captured data as log files.

- F1: log files to be used to store the data captured by P3 and/or P4.

6. **Structure**

   The descriptions of this pattern's structure is that both Px and Fy are deployed in either only the provider side or only the customer side. This does not include deployment of Px in a hypervisor. Figure 4.7 presents this

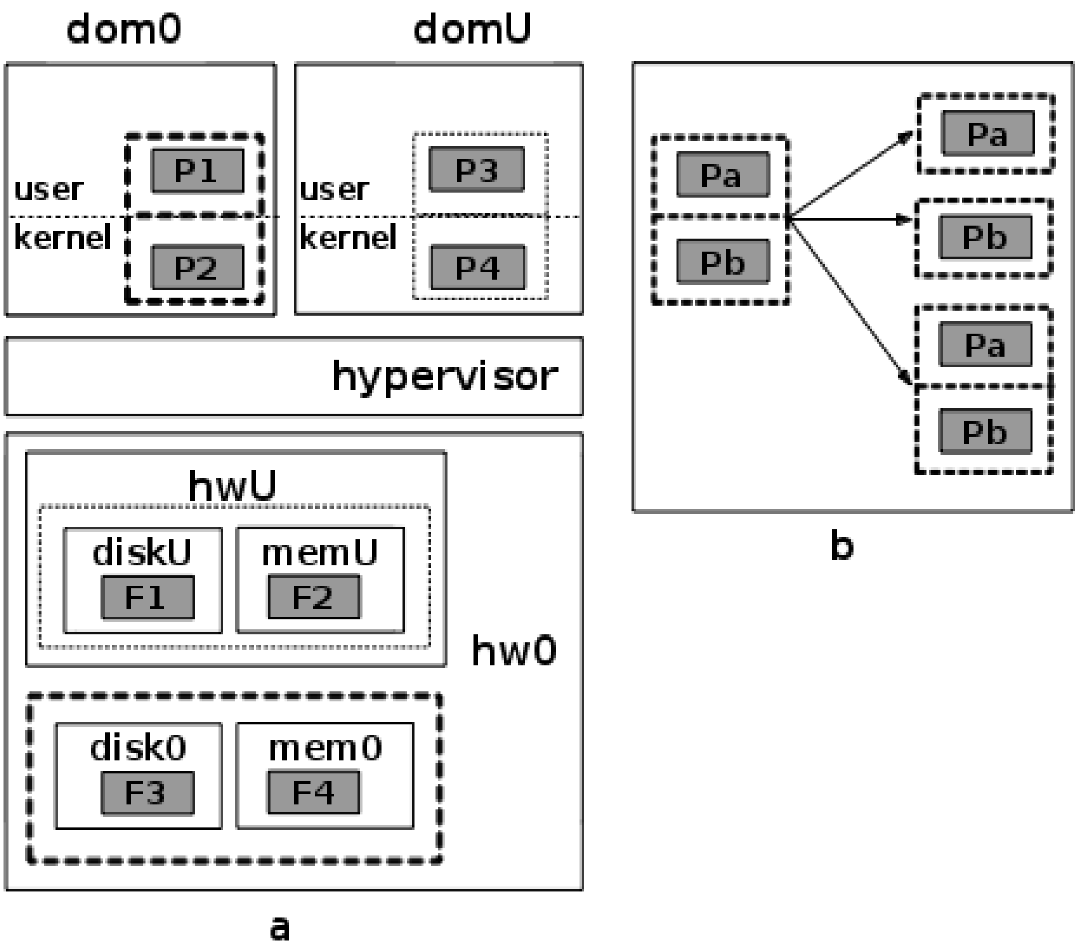


Figure 4.7: A structure of the simple logging system pattern

   structure.

   Figure 4.7 a is an overall structure of the pattern. Figure 4.7 b is expanding descriptions of dotted-line boxes in dom0 and domU. It illustrates all possible forms or architectures when the pattern deploying Pa and/or Pb which creates three architectures. They are: Pa as the architecture that deploys only Pa see the first arrow line from the top of Figure 4.7 b, Pb as the one that deploys only Pb see the second arrow line in the figure, and PaPb as the one that deploys both Pa and Pb see the last arrow line in the figure. For example, when this pattern deploys P1 and/or P2, this creates three architecture as: P1, P2, and P1P2. Figure 4.7 a and b enforce the deployment of an architecture derived from the pattern as discussed in the lists below.

- **The provider side deployment:** when a logging system architecture deploys Px in the provider side (P1 and/or P2 in the boxes in dom0 in Figure 4.7 a), the architecture also deploys Fy in this provider side as F3 and F4 see inside the thick-dot-line box in hw0 in Figure 4.7 a.
- **The customer side deployment:** when a logging system architecture deploys Px in the customer side (P3 and/or P4 or the boxes in domU), then the architecture also deploys Fy in this customer side as F1 and F2 see inside the hwU box in Figure 4.7 a.

Based on the structure of the pattern, there are six concrete logging system architectures based on this pattern. All of them are in the single domain category (Figure 4.1 in Section 4.1.1). From the figure, for the provider side deployment and when following the branches in this figure, there are three architectures. They are the branches of: dom0/P1/disk0, dom0/P2/disk0, and dom0/P1P2/disk0. For the customer side deployment, there are other three architectures: domU/P3/diskU, domU/P4/diskU, and domU/P3P4/diskU.

7. **Collaborations**

   - Provider side deployment
     - P1 can collaborate with P2 (and vice versa) to capture the necessary logging data and to store the captured data as log files or F3.

    - P1, P2, or both normally write(s) the captured data to disk0 as F3.

- Customer side deployment
    - P3 can collaborate with P4 (and vice versa) to capture the necessary logging data and to store the captured data as log files or F1.
    - P3, P4, or both normally write(s) the captured data to diskU as F1.

8. **Consequences**
   Below are benefits and liabilities of this pattern. Basically, a logging system architecture derived from this pattern is simple but not secure.

   - The benefits:
     It is simple, cheap, and fast to design and develop a logging system in IaaS. This is because: there is no unnecessary modifications to a hypervisor; and modifications are in only one side not both sides.
   - The liabilities:
     Security issues of a concrete logging system architecture derived from this pattern come from both sides, as discussed in the lists below.
       - *Security issues when deploying Px in the provider side*
         Section 3.3.3.1 discussed that when a logging system deploys P1 in dom0, which is owned and controlled by a provider, this may lead to integrity issues of the system. This is because malicious insiders may modify P1's code to benefit themselves. This is a serious concern as argued by [46; 88; 89].
       - *Security issues when deploying Fy in the provider side*
         Section 3.3.3.2 discussed that when any logging system deploys F3 in hw0, this may lead to privacy issues. This is because malicious insiders may learn about customers' personal data from F3 or F4.
       - *Security issues when deploying Px in the customer side*

Section 3.3.3.1 discussed that when deploying P3 in domU, this may lead to integrity issues because domU cannot be trusted from the providers' perspective to have installed security software, as argued by [78]. Attackers who compromise and take control over domU or malicious domU owners may modify P3 in domU to alter logging system's behaviour to benefit themselves, as will be fully discussed in Section 5.4 and 5.5. Placing P3 in domU also allows the owner of domU to tamper with the components, as mentioned by [30; 39; 78; 96; 97].

– *Security issues when deploying Fy in the customer side*
  This is the same as the drawback of P3 discussed above. This is because domU cannot be trusted [78]. When a pattern deploys F1 and F2, attackers who compromise and take control over domU or malicious domU owners may modify these files.

9. **Implementation**

Below are some helpful techniques for implementing the pattern.

- As discussed in Section 2.12 the kernel usually resides in an elevated system state compared to normal user applications [2]. This includes a protected memory space and full access to the hardware [2]. Thus, a system should be more robust when deploying Px in a kernel level compared to in user level. Then, for example, it will be more difficult for attackers to compromise Px.

- When a concrete logging system architecture derived from this pattern deploys Px in user level in domU or dom0, the system may be less robust compared to when deploying Px in the kernel. For example, binding Px with the kernel, which is in a protected memory space, is more difficult for attackers to compromise compared to compromising Px in user level.

- Section 3.5.2.5 already and fully discussed flexibility and re-usability of development of Px and Fy. There are existing user level Px such as libVMI [79] (as P1 discussed in Section 3.4.2.1) or the JAR logging

[76] (as P3 discussed in Section 2.8), thus ones can reuse them. Fy can be implemented by re-using existing logging file systems such as a tree-based data structure [83], discussed in Section 3.5.2.2.

10. **Sample Code**
    N/A

11. **Known Uses**
    Below are two existing concrete architectures of this pattern, see Figure 4.1.

    - The dotted-lines labelled one in Figure 4.1 create the architecture of dom0/P1/disk0. Actual systems that apply this architecture include: a network monitoring application in [1]; a demo monitoring program in [39]; and the logging application (Figure 3.3) in the spamming case study in Section 3.4.2.
    - The dotted-lines labelled two in Figure 4.1 create the architecture of domU/P3/diskU. The JAR logging [76] system applies this architecture.

12. **Related Patterns**
    A logging system that is derived from the simple logging system pattern when this system deploys Px and Fy in the provider side can be also considered as a derivation of the security facilitator logging system pattern. This will be fully described in the 'Related Patterns' element of the security facilitator logging system pattern in Section 4.2.2.2.

4.2.2.2 Security Facilitator Logging System Pattern

The security of logging systems themselves is very important. To enforce the security policy of these systems in the IaaS context is also significant. Thus, we separate the security facilitator logging system pattern from the simple logging system pattern. In IaaS context, a concrete logging system architecture that is derived from this pattern can facilitate enforcement of a security policy of the system itself in terms of integrity considerations.

Anderson et al [98] define a security policy as "a high-level specification of the security properties that a given system should possess". Bishop [99] argues that all related features of confidentiality, integrity, and availability are the consideration of a security policy. This section limits the discussions to the integrity consideration of a security policy, i.e., the integrity policies. In the context of logging in IaaS, the security concern is the integrity of log files (Fy) that is produced by Px as discussed in the end of Section 2.11. Additionally, for the integrity of Px that produces Fy, Px must not be randomly changed.

Bishop [99] argues that the difference between commercial requirements and military requirements is that the former focuses on preserving data integrity. As the goals of the integrity policies, he also refers to five requirements which is defined by Lipner [100]. The first requirement is that:

users^{1} *will not* ***write***^{2} *their own* ***programs***^{3}, *but will use* ***existing production programs and databases***^{4}.

To create a requirement of the integrity policies of a logging system, the first requirement by Lipner also can apply to this creating requirement as:

domU owners, or customers$^{1'}$ *will not* ***develop***$^{2'}$ ***logging systems***$^{3'}$, *but will use* ***the logging systems developed by providers***$^{4'}$.

With the different contexts, the ***users***^{1} is similar to the ***domU owners, or customers***$^{1'}$, ***write***^{2} is similar to ***develop***$^{2'}$, and so on.

DomU cannot be trusted from the providers' perspective to install security software (e.g., logging systems) [78] as discussed in Section 3.3.3.1. Concerning our created requirement of the integrity policies of a logging system, when one deploys Px and Fy in domU, an owner of this domU may change these components. If we want to facilitate enforcement of these integrity policies, we need to obey the created requirement. This is achieved if: **Px and Fy are deployed in the provider side**.

In IaaS logging context, this pattern facilitates enforcement of a security policy of a logging system in terms of the integrity aspect discussed above. The elements below are the descriptions of this pattern.

1. **Pattern Name**

   Security Facilitator Logging System

2. **Intent**

   Facilitate enforcement of a security policy of a logging system in terms of the integrity considerations.

3. **Motivation**

   - **Problem**

     Deploying Px and Fy in domU allows the owner of this domU to tamper with them, as agreed by [30; 39; 78; 96; 97]. [97] states that when a monitored machine resides in the same machine as the detector system, the owners of the monitoring machine can disable the detection system, or modify the detected data.

   - **Solution** The solution is to deploy Px and Fy only in the provider side.

   - **How and why the pattern works, and its example scenario**

     An IaaS server has dom0 that is responsible for the administration of domUs [41]. One dom0 in the provider side can host a number of domU in the customer side. The number can be numerous. Ideally, one logging system should monitor all the monitored domUs, not one logging system for one monitored domU.

     Regarding enforcement of a security policy of a logging system in terms of the integrity aspects, deploying a logging system (Px and Fy) in the provider side makes the enforcement easier than deploying the system in the customer side. The first reason is that deploying a logging system in the provider side needs only one logging system in dom0 to monitor all domUs.

     In contrast, deploying logging systems in the customer side needs a logging system in each of all domUs. If the number of domUs is numerous, it should be difficult to enforce the security policy of each of all the systems in all the domUs, compared to enforce the policy of only one system in dom0.

The second reason is that, if deploying logging systems in all domUs instead of deploying one logging system in dom0, this allows the owners of domUs to tamper with the systems, as discussed above. This is a challenge for the enforcement of the policy. Lastly, it should be difficult for domU owners to tamper with (e.g., alter the code of) a logging system in the provider side. This is because the domU owners have no control over the components that are in the provider side. A logging system in the provider side makes it easier for the enforcement of the policy.

4. **Applicability**

   Use the pattern when:

   - To facilitate enforcement of a security policy of a logging system in terms of the integrity aspects in IaaS.
   - A system should be in only the provider side.

5. **Participants**

   - P1, P2, P5: processes that individually or co-operationally capture the necessary logging data and store the captured data as log files.
   - F3: log files of the captured data by P1, P2, and/or P5.

6. **Structure**

   The descriptions of the structure of this pattern is that 'Px and Fy need to be deployed only in the provider side'. Figure 4.8 presents the structure of this pattern. Figure 4.8 a is an overall structure of the pattern. When a logging system deploys P1, P2, and/or P5 (see the dotted-line boxes in dom0 and in hypervisor in Figure 4.8 a), this system also needs to deploy F3 and F4 see the dotted-line box in hw0 in Figure 4.8 a. The deployment of these Px and Fy is in the provider side.

   Figure 4.8 b is expanding descriptions of the dotted-line boxes in dom0 and in hypervisor. It illustrates all possible forms or architectures when

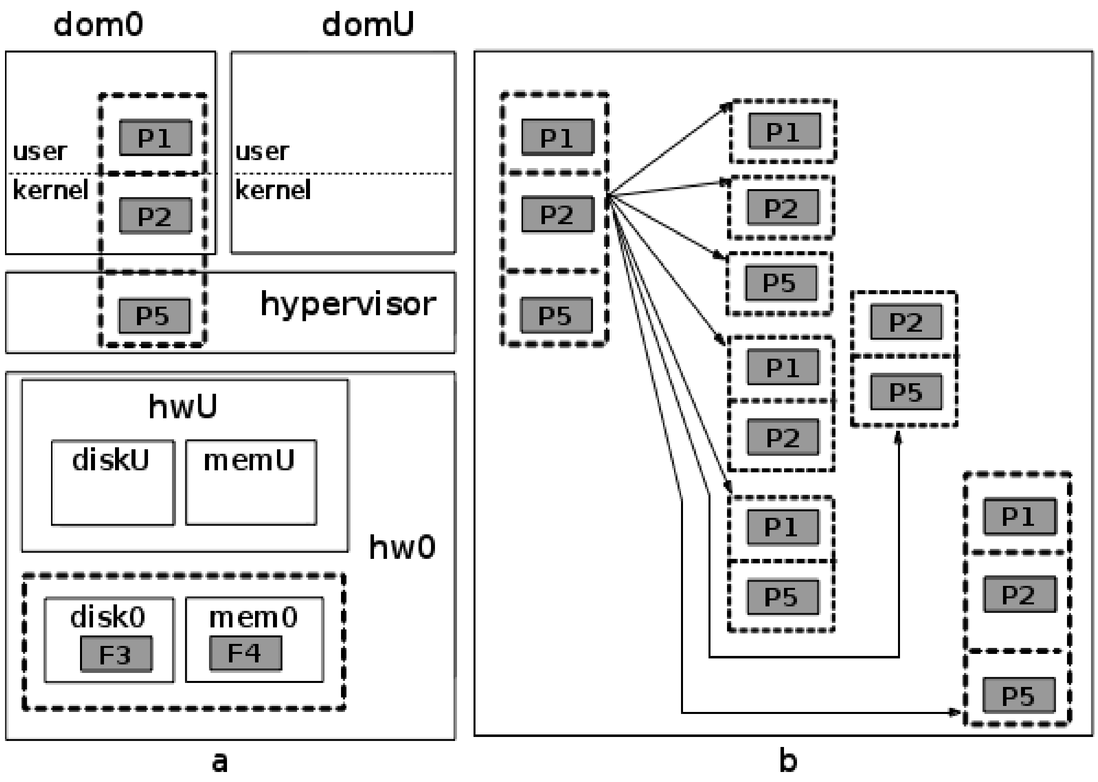


Figure 4.8: A structure of the security facilitator logging system pattern

the pattern deploying P1, P2, and/or P5 which creates seven architectures. They are: P1, P2, P5, P1P2, P1P5, P2P5, and P1P2P5, see the first to the seventh arrow lines from the top of Figure 4.8 b. Then, each architecture needs to deploy F3 and F4.

There are seven concrete architectures of the pattern. In the single domain category or Figure 4.1, for the provider side deployment and when following the branches in this figure, there are four architectures. They are the branches of: dom0/P1/disk0, dom0/P2/disk0, dom0/P1P2/disk0, and hypervisor/P5/disk0. In the two domains category or Figure 4.3, when following the branches in this figure, there are three architectures. They are the branches of: dom0H/P1/P5/disk0, dom0H/P2/P5/disk0, and dom0H/P1P2/P5/disk0.

7. **Collaborations**

   - Px (P1, P2, and/or P5) can collaborate with another or other Px (and vice versa) to capture the necessary logging data and to store the captured data as log files or F3.
   - P1, P2, and/or P5 normally write(s) the captured data to disk0 as F3.

8. **Consequences**
   Below are benefits and liabilities of the pattern. In summary, this pattern facilitates enforcement of a security policy of a logging system in terms of the integrity aspects, or for short: the integrity policies of a logging system. However, when deploying Px and Fy in provider sides, this may give them more control over the logging systems.

   - The benefits:
     - It facilitates enforcement of the integrity policies of a logging system.
     - This pattern could benefit rapid logging system development which is an important aspect for the development of logging systems in the cloud, as discussed in Section 3.5.2.6. This is because there will be modifications on only provider components (e.g., dom0

  and/or hypervisor) but domUs in the customer side, which can be a numerous number of them.

  - The next advantage is small performance impact on domUs. This is because there are no modifications to domUs in the customer side.
  - When a logging system is in the provider side, it should be difficult for malicious domU's owners from the customer side to tamper with (e.g., alter the code of) Px and/or Fy in the system.
  - This thesis focuses on the systematic aspects for accountability in the cloud as discussed in Section 1.1. One of the aspects is to concern reducing a TCB size of a logging system. The TCB is discussed in Section 1.1, 1.2, 2.13, and 3.5.2.8. A logging system derived from this pattern yields the smallest TCB size compared to the TCB sizes yielded from the other patterns. Section 3.5.2.8 has discussed the smallest TCB size of a logging system (Figure 3.7).
  - A logging system derived from this pattern can simplify the management of the system. This is because the management will be achieved in only the provider side, not in both the provider side and the customer side.
  - As discussed in Section 3.3.3.1 that domU in the customer side can not be trusted to deploy security software inside it. This pattern has no deployment of Px and Fy in customer sides. Thus, there are no security issues generated from the customer side.

- The liabilities:
  - *More control over logging systems by a provider*
    When deploying Px and Fy in the provider side, this may give it more control over a logging system. This leads to security issues discussed below.
  - *Security issues when deploying Px and Fy in the provider side*
    Section 3.3.3.1 discussed that when a logging system deploys P1 in the provider side or in dom0, this may lead to integrity issues of the system. This is because malicious insiders may modify P1's

code to benefit themselves. This is a serious concern as argued by [46; 88; 89].

Section 3.3.3.2 discussed that when any logging system deploys F3 in hw0, this may lead to privacy issues. This is because malicious insiders may learn about customers' personal data from F3 or F4.

9. **Implementation**

Below are some helpful techniques for implementing the pattern.

- When a concrete logging system architecture that is derived from this pattern deploys Px in only a hypervisor, the system architecture yields the smallest TCB size (Figure 3.7).
- As discussed in Section 2.2.1 that a hypervisor is run directly on hardware of a computer machine. It allows the machine to run multiple guest OSs (dom0 and a number of domUs) at the same time [52]. Thus, regardless of the hardware, a hypervisor is in a lower layer compared to dom0.

  A system should be more robust when deploying Px in a hypervisor level compared to when this Px is in a dom0 user or kernel level. This is because a dom0 user or kernel level is in a higher layer compared to a hypervisor. Thus, binding Px with a hypervisor is more difficult for attackers to compromise the Px in a hypervisor level compared to compromising the Px in a dom0 user or kernel level.

  However, to place Px in a hypervisor is a challenge. One of the reasons can be that the source code of the present hypervisors including VMware and Xen are large and complex [95].
- Section 3.5.2.5 discussed re-usability of Px and Fy. There are existing Px that can work in the provider side such as libVMI [79] (as P1, discussed in Section 3.4.2.1), thus a logging system that is derived from this pattern can reuse this existing Px. The system can implement Fy by re-using existing logging file systems such as a tree-based data structure [83], discussed in Section 3.5.2.2.

10. **Sample Code**
N/A

11. **Known Uses**
Below are two existing concrete architectures derived from this pattern, see Figure 4.1.

    - The dotted-lines labelled one in Figure 4.1 create the architecture of dom0/P1/disk0. Actual systems that use this architecture include: a network monitoring application in [1], a demo monitoring program in [39], and the logging application (Figure 3.3) in the spamming case study in Section 3.4.2.
    - The dotted-lines labelled three in Figure 4.1 create the architecture of hypervisor/P5/disk0. This architecture is for AVMs [44] system and a system in [93].

12. **Related Patterns**
A logging system that is derived from this pattern when the system deploys Px and Fy in the provider side without deploying P5 is also considered as a derivation of the simple logging system pattern. In this case, this system inherits main benefits from both patterns. They are to facilitate enforcement of a security policy of the system itself and to enable simple and cheap design and development of the system.

4.2.2.3 Complex Logging System Pattern

The pattern is capable of capturing log files across the customer side (domUs) and the provider side (a hypervisor and dom0). Examples of the log files are discussed by Ko et al [30]. They explain that when one domU (that is hosted by dom0) sends a file to another domU, the loggers need to record this sending event as a log file of the first domU (called a log fileA). At the same time and to correspond to log fileA, the corresponding dom0 log file (called log fileB) is also generated in this dom0. Log files across the customer side (domUs) and the provider side (dom0) can be used to provide full transparency of all operations in the cloud [30].

1. **Pattern Name**
   Complex Logging System

2. **Intent**
   Provide abilities to capture the necessary logging data across the customer side and the provider side.

3. **Motivation**

   - **Problem**

     For loggers[1] to perform effective monitoring of data in the cloud, Ko et al [30] introduce five necessary requirements. One of them is 'transcend virtual machine (VM) and physical machine (PM)[2]'. To do so, their loggers must be in kernel space, and must be able to transcend both domU and dom0 spaces. This provides full transparency of all operations in the cloud [30]. To satisfy this requirement, their loggers need to have abilities to capture the necessary logging data across the customer side (domUs) and the provider side (dom0s).

   - **Solution**

     To provide such abilities to capture the necessary logging data across domUs and dom0s, Ko et al [30] deploy their Px and Fy in both the customer side and provider side. We believe that deploying Px and Fy in both sides can provide such abilities to logging systems as done by [30]. This provision is significant for logging and accountability approaches in a complicated IaaS environment which involves virtualisation infrastructure and more than one party. Note that, this chapter does not investigate capturing the necessary logging data as log files across both sides by deploying Px and Fy only in either sides.

   - **How and why the pattern works, and its example scenario**

     To provide full transparency of all operations in the complicated IaaS environment and to benefits the accountability, logging systems should

[1] They are Px in our generic logging components (Figure 3.1)
[2] A PM is dom0 in our IaaS architecture in Figure 2.1 and 3.1

be able to record as much necessary logging data as possible. The systems have to be equipped with sophisticated abilities such as to capture the necessary logging data across domUs and dom0s as discussed above. Deploying Px and Fy across both customer and provider sides means that Px and Fy can be distributed to all critical inner layers of the IaaS infrastructure such as in dom0 user level or domU kernel. This should enable Px and Fy to cooperate in capturing of the necessary logging data as much as possible.

4. **Applicability**
Use the pattern when:

   - To provide a logging system in IaaS infrastructure with concerning to cover abilities to capture the necessary logging data across both a customer and provider side.
   - Px and Fy of a system should be in both a customer and provider side.

5. **Participants**

   - Provider side participants
     - P1, P2, P5: processes that individually or co-operationally capture the necessary logging data and store the captured data as log files.
     - F3: log files of the captured data by P1, P2, and/or P5.
   - Customer side participants
     - P3, P4: processes that individually or co-operationally capture the necessary logging data and store the captured data as log files.
     - F1: log files of the captured data by P3 and/or P4.

6. **Structure**
The descriptions of this pattern's structure is that a concrete logging system architecture derived from this pattern has to deploy Px and Fy in both customer and provider sides. The descriptions can be interpreted as: this system architecture needs to deploys at least one Px in the provider side

and at least another Px in the customer side. It then has to deploy F1 in the customer side and F3 in the provider side.

Figure 4.9 presents the structure of this pattern. A concrete logging system

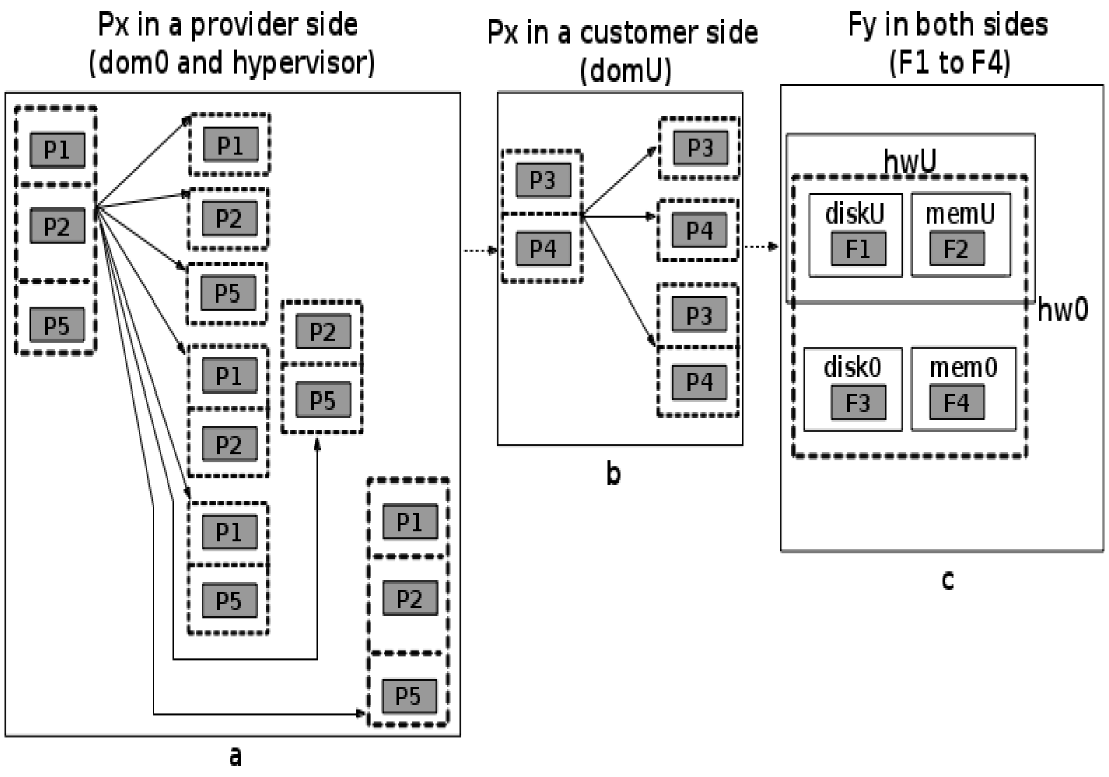


Figure 4.9: A structure of the complex logging system pattern

architecture that derived from this pattern has to follow the structure in this figure. From the figure, the system architecture needs to deploy at least one Px in the provider side or choose one of the seven forms pointed by the arrow lines in Figure 4.9 a. Then, it needs to deploy at least one Px in the customer side or choose one of the three forms pointed by the arrow lines in Figure 4.9 b. Finally, the architecture has to deploy Fy in both sides see Figure 4.9 c which is deployment of F1 and F2 in the customer side (hwU) and F3 and F4 in the provider side (hw0).

There are 21 concrete logging system architectures based on this pattern. First 12 architectures are from the two domains category or Figure 4.3.

From the figure, all the concrete architectures can be created by following any branch of dom0U or domUH then ending with disk0U. For example, the dotted-lines labelled one in the figure create the branch or of dom0U/P2/P4/disk0U concrete architecture.

Other 9 architectures are from the three domains category or Figure 4.6. From the figure, all the concrete architectures can be created by following any branch of dom0UH/P1, of dom0UH/P2, or of dom0UH/P1P2 then ending with disk0U. For example, the dotted-lines labelled one in the figure create the branch of dom0UH/P1/P3/P5/disk0U concrete architecture.

7. **Collaborations**

    - The collaborations within provider side participants
        - Px (P1, P2, P5) can collaborate with another or other Px (and vice versa) to capture the necessary logging data and to store the captured data as log files or F3.
        - P1, P2, and/or P5 normally write(s) the captured data to disk0 as F3.
    - The collaborations within customer side participants
        - P3 can collaborate with P4 (and vice versa) to capture the necessary logging data and to store the captured data as log files or F1.
        - P3, P4, or both normally write(s) the captured data to diskU as F1.
    - Provider side participants in collaboration with customer side participants
        - The participants of each side can collaborate with the participants of another side (and vice versa) for capturing and storing the necessary logging data.

8. **Consequences**

    Below are benefits and liabilities of the pattern.

- The benefits:
  - It provides abilities to capture the necessary logging data across the customer side and provider side.
  - It is not only either the customer side or provider side to completely control over logging systems. When deploying Px and Fy in both sides, this may give both sides to share the control over the logging systems.
- The liabilities:
  - It may be difficult to enforce a security policy of a logging system. This is because Px and Fy are distributed across both sides.
  - It may not be cheap and fast to design and develop logging systems derived from this pattern. This is because both Px and Fy are distributed into both sides.
  - A system derived from this pattern yields the biggest TCB size compared to the TCB sizes yielded from the other patterns. For this pattern, this is because a system's TCB size needs to include domU from the customer side. Section 3.5.2.8 has discussed the biggest TCB size of a logging system see Figure 3.6.
  - Security issues of a logging system can be generated from both sides at the same time. This should be difficult to mitigate these issues compared to the issues generated from only either side.
    - *Security issues when deploying Px in the provider side*
      Section 3.3.3.1 discussed that when a logging system deploys Px such as P1 in dom0 at the provider side, this may lead to integrity issues of the system. This is because malicious insiders may modify Px's code to benefit themselves. This is a serious concern as argued by [46; 88; 89].
    - *Security issues when deploying Fy in the provider side*
      Section 3.3.3.2 discussed that when any logging system deploys F3 in hw0, this may lead to privacy issues. This is because malicious insiders may learn about customers' personal data from F3 or F4.

- *Security issues when deploying Px in the customer side*
  This is because, for example, Section 3.3.3.1 discussed that when deploying Px such as P3 in domU, this may lead to integrity issues because domU cannot be trusted from the providers' perspective to have installed security software, as argued by [78].
- *Security issues when deploying Fy in the customer side*
  This is the same as the drawback of Px in the customer side discussed above. This is because domU cannot be trusted [78]. When a pattern deploys F1 and F2, attackers who compromise and take control over domU or malicious domU owners may modify these files.

9. **Implementation**

   Below are some helpful techniques for implementing the pattern.

   (a) The smallest TCB size of a concreted system derived from this pattern is when the system does not deploy P3 in domU.

   (b) The robustness of a system can be improved when there is no Px deployed in the user levels of domU and dom0.

   (c) Ones can reuse existing Px such as libVMI [79] (as provider side Px) or the JAR logging [76] (as customer side Px). Fy can be implemented by re-using existing logging file systems such as a tree-based data structure [83] (as provider or customer side Fy).

10. **Sample Code**
    N/A

11. **Known Uses**

    We found only one existing concrete architecture of this pattern, see Figure 4.3. From the figure, the dotted-lines labelled one create the branch of dom0U/P2/P4/disk0U. This is architecture of Flogger [30] presented in Figure 4.4.

12. **Related Patterns**
    N/A

4.2.3 Conclusions on the advantages and disadvantages of the patterns

This section gives conclusions on the advantages and disadvantages of the patterns.

Although the advantages and disadvantages were discussed in the 'consequences' element of each pattern, this section draws conclusions on the advantages and disadvantages again based on typical good programming guidelines or good security guidelines for design a robust monitoring architecture discussed by Payne et al [39].

Payne et al [39] design a robust monitoring architecture based on six high-level requirements. They identify these requirements and argue that the requirements can be considered as typical good programming guidelines or good security guidelines. They also state that some of their identified requirements could be recognized as specially designed versions of classic security design principles proposed by Saltzer and Schroeder [101].

Although their requirements are for designing a monitoring architecture, we believe they apply to logging architectures as well. This is because the monitoring and the logging system architectures in an IaaS are the same except the former architectures do not need to have log files while the latter architectures do, as discussed in Section 2.9. However, only four of their requirements can apply to our logging context. When applying these four requirements to our logging approaches, we paraphrase some of their technical terms in our context. Our four paraphrased requirements are: no unnecessary modifications to a hypervisor, no modifications to domUs, small performance impact on domUs, and the rule that domU can not tamper with (e.g., alter the code of) Px and/or Fy in a logging system.

This section uses these four paraphrased requirements as the advantages and disadvantages of a logging system derived from a particular pattern. When the system satisfies our requirements, then these requirements can be seen as the system's advantages. Otherwise, they can be considered as the system's disadvantages. We also add three requirements from the main benefits of the simple logging system, security facilitator logging system, and complex logging system pattern. They are a simple logging system, to facilitate enforcement of a security policy of a logging system, and abilities to capture the necessary logging data across the customer side and the provider side. These additional requirements also are significant for comprehending the benefits and liabilities of applying the patterns and for assessing them.

We omit two of the six high-level requirements discussed by Payne et al [39]. The first one is rapid development of new logging systems. To satisfy this requirement, a new system should provide APIs that makes the developers' tasks easier. However, in our context, we investigate only how to build logging systems, not yet the system's APIs which could be achieved in the future. The second omitted requirement is abilities to record any data on domUs. This data includes memory and disk I/O, network traffic, CPU context, and static disk contents. However, we discuss to record only behaviour or activities of a process or processes in domU or a domU file's life cycle.

Table 4.1 presents the advantages and disadvantages of the three patterns. In the table, '/' means that a system derived from a particular pattern can satisfy a requirement, and 'X' is the opposite. '/X' means that the system can achieve a requirement only if it also satisfies a condition or conditions in the parenthesis or '()'. Note that, all three patterns have the same common aim which is capturing the necessary logging data. Section 4.2.2 defined this data as: behaviour or activities of a process or processes in domU or as: a domU file's life cycle. Thus, this section gives the conclusions based on only the seven advantages or the requirements in the table.

From the table, the second and the third column are for the simple logging system pattern. The forth and the fifth column are for the security facilitator logging system and the complex logging system pattern respectively. A logging system architecture derived from the simple pattern which deploys Px and Fy only in the customer side (column 2 in the table) can satisfy only requirement 1 and 5 which are: no unnecessary modifications to a hypervisor, and the system is simple and cheap for the design and development respectively.

Requirements	***Simple logging system pattern*** in the customer side	***Simple logging system pattern*** in the provider side	Security facilitator logging system pattern	Complex logging system pattern
1) no unnecessary modifications to a hypervisor	/	/	/X (when a system does not deploy P5)	/X (when a system does not deploy P5)
2) no modifications to domUs	X	/	/	X
3) small performance impact on domUs	X	/	/	X
4) the rule that domU can not tamper with (e.g., alter the code of) Px and/or Fy in a logging system.	X	/	/	X
5) a simple logging system	/	/	/X (when a system does not deploy P5)	X
6) facilitate enforcement of a security policy of a logging system	X	/	/	X
7) abilities to capture the necessary logging data across the customer side and the provider side	X	X	X	/

Table 4.1: The advantages and disadvantages of the three patterns

Whereas, an architecture derived from the simple pattern which deploys Px and Fy only in the provider side (column 3 in the table) cannot satisfy only requirement 7 which is: abilities to capture the necessary logging data across the customer side and the provider side. See column 3 at the row of requirement 6, interestingly, this architecture is also considered as a derivation of the security facilitator logging system pattern. In this case, the architecture is simple and cheap for design and development (the main benefit of the simple logging system pattern), and also can facilitate enforcement of a security policy of the system itself (the main benefit of the security facilitator logging system). The 'Related Patterns' element of the simple and security facilitator logging system pattern in Section 4.2.2.1 and 4.2.2.2 discussed this case.

In the forth column, the architecture derived from the security facilitator logging system pattern cannot satisfy only requirement 7. However, the architecture can satisfy requirement 1 and 5 only when it does not deploy P5. In the last column, the architecture derived from the complex logging system pattern can satisfy only requirements 1 and 7, and it can satisfy requirement 1 only when it does not deploy P5.

4.3 Analysis of existing logging and monitoring works and our proposed system in relation to the patterns

This section analyses existing logging and monitoring works and our proposed system in relation to the patterns. To do so, it discusses advantages and disadvantages or characteristics of three existing logging systems, our proposed system, and two existing monitoring systems. We use the conclusions of the three patterns in Table 4.1 as tools to analyse all the existing work architectures in relation to the patterns.

The goal of this section is to leverage our patterns in order to give examples for analysing the characteristics of logging systems which are derived from such patterns. The discussions in this section could be sophisticated examples or case studies to illustrate and evaluate the patterns.

Section 2.8, 2.9, and 3.3.2 already gave the details of the existing logging systems in the related work. Each related work may be modified to be able to achieve the necessary logging data. This section focuses on analysing these existing logging systems in relation to the patterns rather than to explain the details of all the components of the system.

This section draws conclusions on the advantages and disadvantages encountered when using existing logging systems that are derived from different patterns. The result is a spectrum of patterns for describing how to construct logging systems of varying characteristics. The characteristics of a logging system are based on the combination of advantages and disadvantages of its pattern. To introduce examples of the spectrum, in the following sections, we discuss advantages and disadvantages of three existing logging systems, our proposed system, and two existing monitoring systems.

We could not absolutely conclude that, in terms of the characteristics of logging systems that are derived from our patterns, which architecture or pattern is better than which one, or which architecture or pattern is the best one. However, there is a variety of architectures and patterns that can be chosen by the participated parties to suit their needs in building logging systems in an IaaS cloud. Based on the current deployment of Px and Fy of some existing system architectures from the previous work such as PASSXen [43] in Figure 4.5 or Lares [77] in Figure 4.10 (assuming Lares deploys F3 for this discussion), they cannot be considered as architectures derived from our patterns. However, they may be modified to do so. This is out of the scope of this thesis.

The following subsections below analyse all the existing work architectures in relation to the three patterns. Regarding the current deployment of Px and Fy of the existing work architectures, each of the architectures is considered as a derivation of one of the three patterns. We use the conclusions of the three patterns in Table 4.1 as tools to analyse all the existing work architectures in relation to the patterns. The analysis reveals advantages and disadvantages or characteristics of all the architectures.

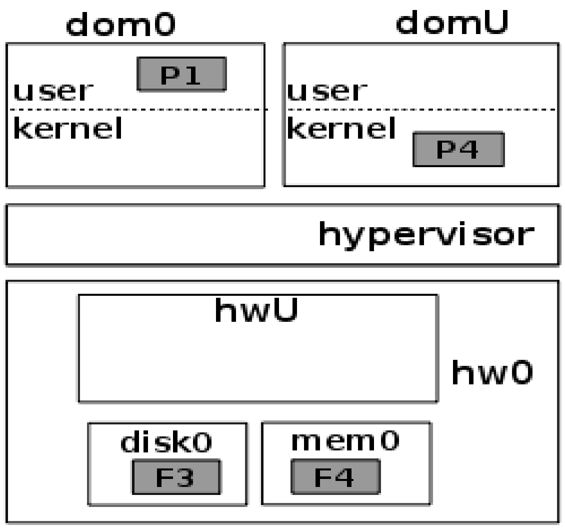


Figure 4.10: An architecture of Lares

4.3.1 Analysis of existing logging works

Section 2.8 investigated previous research that can be considered as work that could be applied in a logging approach in IaaS. This subsection analyses the architectures of logging systems in the previous research in relation to the three patterns. Based on the current Px and Fy deployment of the architectures, each of them is considered as a derivation of one of the three patterns. This subsections uses the conclusions of the three patterns in Table 4.1 as tools to analyse all the architectures in relation to the patterns, below.

- **The JAR logging [76]**

  In this work, log files have to be strongly tied to a user's data file or corresponding user's JAR file [76]. Thus, to apply this work in an IaaS environment, a user's data file (which also includes log files) is stored in diskU. When this data file is activated, its logging mechanism as P3 is also activated. Then, this mechanism performs logging tasks in domU user level, and the log files that are produced by this mechanism are stored in diskU as F1.

  The system architecture of this work (Figure 4.11) deploys three critical

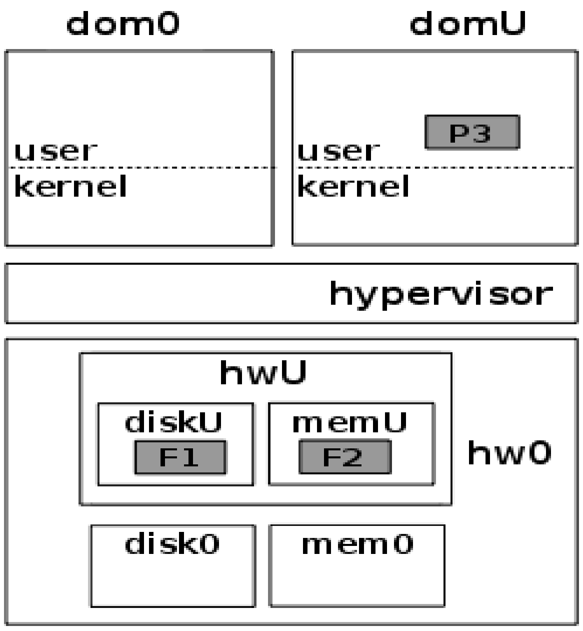


Figure 4.11: The system architecture of Jar logging: a derivation of the simple logging system pattern when the architecture deploying Px and Fy in the customer side.

logging components: one logging processes component which is P3, and two log file components including F1 and F2. This system architecture can be considered as a derivation of the simple logging system pattern with the architecture deploying Px and Fy in the customer side.

From the conclusions of the simple logging system pattern when Px and Fy deployed in the customer side in the Table 4.1, this architecture is quick, cheap, and simple ways to design and develop logging systems in an IaaS. Moreover, it has no modifications to a hypervisor.

- **AVMs [44]**

  The architecture of this work can be modified to record the necessary log data. The main process of this work is a logger as P5 that resides in a hypervisor to record incoming and outgoing network packets of domUs. The logger then permanently stores the domU intercepted network data as F3 in disk0. The architecture of AVMs is Figure 4.12. This system deploys

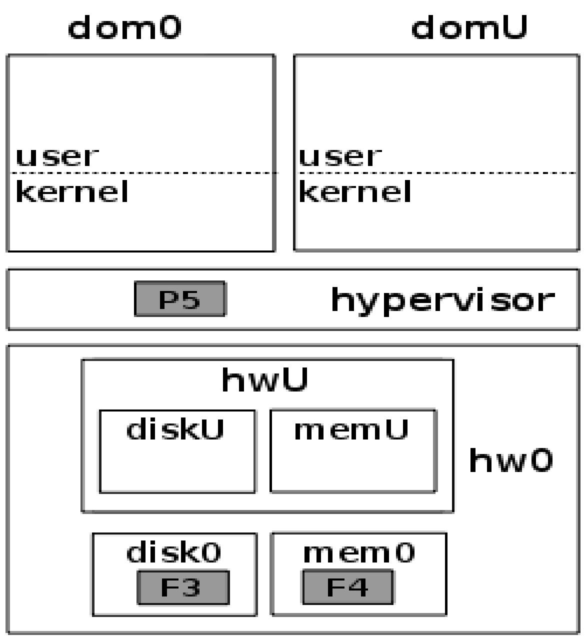


Figure 4.12: The system architecture of AVMs: a derivation of the security facilitator logging system pattern

three critical components: one logging processes component which is P5; and two log file components include F3 and F4.

From the conclusions of the security facilitator logging system pattern in Table 4.1, AVMs have no abilities to capture the necessary logging data across the customer side and the provider side. This architecture is not simple because the needs to modify a hypervisor.

- **Flogger [30]**

  Section 3.4.1 discussed the main components of Flogger in details. A pattern of this system is Figure 4.4. Flogger deploys six critical logging components: two logging processes components which are P2 and P4; and four log file components including F1 to F4. The system is a derivation of the complex logging system pattern.

  From the conclusions of the complex pattern in Table 4.1, Flogger has no unnecessary modifications to a hypervisor. It has abilities to capture the necessary logging data across the customer side and a provider side.

4.3.2 Analysis of our proposed system

The architecture in Figure 4.2 is for our proposed logging system in the spamming case study in Section 3.4.2. This section discusses characteristics or advantages and disadvantages of this architecture. From the figure, this architecture deploys three critical components: one logging processes component which is P1; and two log file components include F1 and F2. The architecture is a derivation of the simple logging system pattern when the system deploys Px and Fy in the provider side, without deploying P5.

This makes the architecture is also considered as a derivation of the security facilitator logging system pattern. From the conclusions in Table 4.1, this architecture is simple and cheap ways for design and development the system, and it also facilitates enforcement of a security policy of the system itself.

4.3.3 Analysis of existing monitoring works

We investigated previous research of monitoring in the cloud which can be considered as work that could be applied in a logging approach in IaaS. As discussed in Section 2.9, monitoring work does not deploy log files or Fy. For demonstration of analysing these work, we assume that they deploy F3. These works below could be modified to perform logging tasks. They all deploy libVMI, as previously known as XenAccess [39].

- **A network monitoring application in [1]**

  This application is comprised of many modules (including XenAccess) as P1 only in dom0 user level. The details of the modules can be found in [1]. A pattern of this application is Figure 4.13.

- **A demo monitoring program in [39]**

  The program needs XenAccess as P1 in only dom0 user level. The details can be found in [39]. Thus, a pattern of this program deploys only P1,

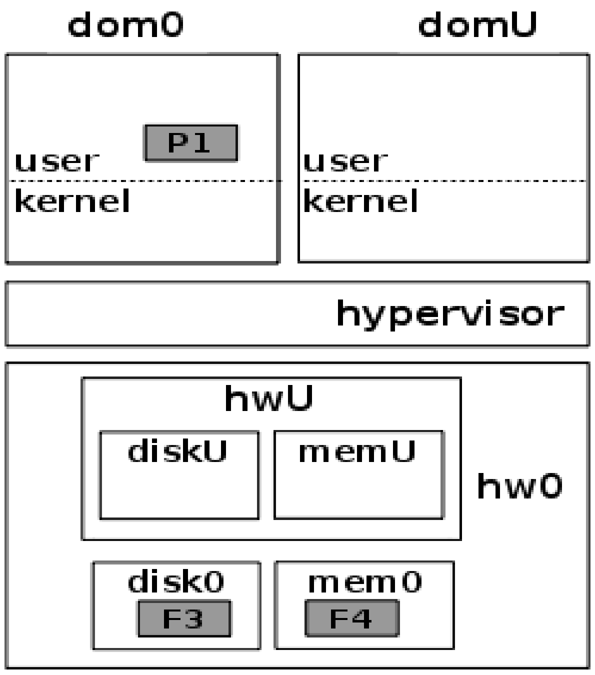


Figure 4.13: A pattern of a network monitoring application in [1]

and it is the same as the pattern of a network monitoring application in [1] above which is Figure 4.13.

Architectures of both works are the same as our proposed architecture above. Their characteristics can be considered as the same as ours.

4.4 Conclusions

This chapter mainly addresses research Gap 4, which is the lack of descriptions of logging systems in terms of design patterns of the systems' components. Its main contribution is three proposed design patterns in the context of logging in IaaS cloud.

The proposed patterns can bring a number of benefits as the same as benefits from design patterns in object-oriented software design and development area. Thus, the proposed patterns could increase reusability of the design and development of logging systems. Moreover, designers should access the proposed patterns more easily. Additionally, the proposed patterns could assist a designer adopts design approaches which make a logging system reusable and not to choose approaches which do not concern reusability concepts. Lastly, they proposed patterns can also enhance the documentation and maintenance of existing logging systems.

We provide a spectrum of patterns for describing how to construct logging systems with varying characteristics. For developers, when building a logging system, the knowledge of characteristics of this system could assist them to get the right design of the system with minimal effort and time commitments. To the best of our knowledge, these three logging patterns are not yet described in the literature.

This chapter describes the related work in the form of patterns. These patterns can be made up of the generic logging components in Figure 3.1 or can be associated to the logging component configurations. Then, the chapter draws conclusions on the advantages and disadvantages when using different patterns. This chapter can be used to clarify the fact that a number of patterns and logging system architectures based on these patterns are missing, for example, our proposed logging system in the spamming case study in Section 3.4.2.

This system is based on a pattern that is quit specifically more easily to be deployed, but it is not very robust. In addition to the spectrum of patterns, we also present sophisticated examples or case studies to illustrate and evaluate the proposed patterns.

In the next chapter, the generic logging components from Chapter 3 are utilised as the basis for an analysis of how real world threats, specifically CSA threat 1, affect both the customer and provider simultaneously. These components are also used as the basis for design of the proposed logging solutions in mitigating the risks associated with threat 1, in order to benefit both the customer and provider sides.

The architecture of our proposed logging system in the spamming case study in Section 3.4.2 is also used in Chapter 6 as the basis for design of the proposed logging solutions in mitigating the risks associated with threat 1, in order to benefit both the customer and provider sides. This is because the architecture is simple and cheap ways for the design and development (the main benefit of the simple logging system pattern in Figure 4.7 Section 4.2.2.1) as discussed in Section 4.2.3. The architecture also facilitates enforcement of a security policy of the system itself, which is the main benefit of the security facilitator logging system pattern or Figure 4.8 in Section 4.2.2.2.

Chapter 5

Logging Solutions

In this chapter, the generic logging components from Chapter 3 are utilised as the basis for an analysis of how real world threats, specifically CSA threat 1, affect both the customer and provider simultaneously. These components are also used as the basis for design of the proposed logging solutions in mitigating the risks associated with threat 1, in order to benefit both the customer and provider sides.

This chapter mainly addresses research Gap 2 which is the issue that research relating to logging in IaaS only focuses on system-centric logs. It also addresses research Gap 1 (a) which is the apparent lack of simultaneous consideration of both the customer and provider sides.

This chapter provides two main contributions. The first contribution is the proposed solution to collect file-centric logs rather than system-centric logs. This contribution addresses research Gap 2. There are some logging solutions that emphasise file-centric logs with an interception approach. However, the prototype implementation of the proposed logging system (in the next chapter) can be an alternative approach to collecting file-centric logs to enhance accountability in IaaS. This approach advocates the introspection of customer VM's memory from dom0. The introspection traverses the kernel data structures in the memory.

The prototype implementation of the proposed logging solutions (in the next chapter) can collect the file-centric log history of customers' critical files. The history information is classified as file-centric logs rather than system-centric logs. The file-centric logs can present associations between processes and files in a customer VM, for example, a record of a process P which reads file F. The proposed log files differ from previous work which focus only on system-centric logs such as the connection topology, bus speeds, and processor loads.

The next contribution is the analysis of how CSA threat 1 affects both customers and providers simultaneously, and the proposed logging solutions that assist in mitigating the risks associated with the threat for both customers and providers. This contribution addresses research Gap 1 (a). The analysis illustrates how CSA threat 1 such as mis-usage of customer VMs can affect both sides.

This analysis differs from previous work which usually concerns the effects of the threats to only either the customer side or the provider side, providing solutions for either side, but not for both. The value of the combined analysis is to provide a basis to understand what the contents are that logging solutions need to collect to be used as evidence to deal with threat 1 for both customers and providers.

The remainder of this chapter is structured as follows. Section 5.1 explains the users environment inside a domU and types of domUs. Section 5.2 defines customers' critical files, explains the files' location in the generic logging components, and gives examples of the files. Then, Section 5.3 discusses how to measure the costs of threat 1 which affects both customers and providers, based on the costs of cybercrime.

Section 5.4 provides an in-depth discussion of the effects on customers due to threat 1. The discussion includes: customer's critical files at risk due to threat 1; domU attacks domU; and effects on customers including direct losses and indirect losses when customers' critical files are attacked. Section 5.5 provides an in-depth discussion of the effects on providers due to threat 1. The discussion includes three main aspects. First is that hypervisor or dom0 vulnerabilities may be channels for domUs to attack the provider's cloud infrastructure. Second is that because of these vulnerabilities, it is possible that domU may attack dom0 and uses this dom0 to attack other domUs. Third is a discussion of direct losses and indirect losses when providers are attacked by domUs.

After that, Section 5.6 provides: a definition for the history of a critical file, and examples of the history information, and a discussion of how the history of critical files can be useful for both customers and providers. Following this, there is discussion of: a process behaviour log and the log's examples; and how the process behaviour logs can be useful to deal with spam activities in domUs. After that, it discusses how the process behaviour logs can be useful to enhance the flexibility of auditing in the cloud using logging systems, and how both the history of critical files and process behaviour logs can provide indirect benefits for providers.

Section 5.7 discusses an approach (called **the domU memory introspection approach**, for short: memory introspection) to obtain the history of critical files. This section describes the memory introspection approaches to obtain history of critical files, and discusses how the memory introspection can obtain the history of critical files. It then, from Linux kernel's data structures, explains how to obtain the full path name of a critical file, the operation's name of an operation that is performed on a critical file, and the last accessed and modified times of a critical file. After that, it describes how to obtain creation and deletion records of a critical file. Lastly, it discusses how to obtain process behaviour logs through the introspection approach.

Section 5.8 discusses another approach to obtain the history of critical files, which is **the domU system calls interception approach**, for short: interception. Section 5.9 discusses some aspects of both the introspection and interception approaches. This includes: security aspects of introspection and interception, which need to be analysed; helper components in introspection and interception approaches; the combination of introspection and interception; ideal logging solution regarding the smallest TCB to produce the history of a critical file; and failure of the implementation for the ideal solution.

Section 5.10 discusses a security analysis of the system architecture of the introspection approach which is used as the architecture of the proposed logging system in the next chapter. Finally, the chapter is briefly summarised and concluded in Section 5.11.

5.1 Users Environment Inside A DomU And Its Types

This section explains the users environment inside a domU and the different types of domUs. This section also provides the context of the activities of users inside domUs. These activities can be malicious as caused by threat 1. A domU that is referred to in this section is a Linux-based OS. This OS is the domU in the generic logging components of IaaS cloud in Figure 3.1 Section 3.3.

5.1.1 Root And Standard Users In A DomU

This subsection defines users and user accounts in Linux domUs. In Linux systems, a user can be either people or accounts, which exist for specific applications to use [102]. Each user is identified by a unique integer or user id, and a separate database outside the kernel assigns a text name or user name to each user id [103]. Each user has an account which combines all the files, resources, and information belonging to this user [103]. For discussion in this thesis, we define a user as a person who can use a domU. We define a user account as composed of two pieces of information: i) a user name and ii) a user password. For example, Alice's user account which belongs to Alice may compose of a user name as 'alice' and a user or Alice's password as 'aliceLoginPass'.

A Linux system has two types of users, a root or administrator user and a standard user. A Linux root user has the same privilege as a Microsoft Windows administrator [104] which has complete control over the system. Thus, the root is a privileged user who can access all parts of the system, or execute administrative tasks [105]. More importantly, they have full access to all files in the system. A root user is created during the system installation phase when an installer needs to specify a root password. We use 'Alice' as an example name of a person, and 'alice' as an example user name of Alice in a Linux system for this thesis. This condition applies for every person name and every user name.

For a Linux-based OS such as Fedora [106], a standard user is a user who carries out ordinary tasks not administrative tasks, and can only access her own files not other users' files in the same system [105]. The first standard user is created during the first boot after the system installation. For example, the installer above needs to create her own user name such as 'alice' with the password (that is different from the root password) for alice user during the first boot. Therefore, Alice is a root and also the first standard user known as 'alice'. Alice could add new standard users to this system.

5.1.2 Types Of DomUs In this thesis

This subsection defines types of domUs based on how these domUs are used. The different types will be used in discussions throughout this thesis. Linux is a multi-user system [105] where more than one users share the system. In this study, a domU is a Fedora system. Although linux is a multi-user system, a domU can be configured as a single user or as a multiple users system. For EC2 [17], an 'EC2 instance' which is a running virtual CPU [107] is actually a domU. An EC2 instance can be shared with more than one user [18; 108]. Thus, due to the sharing of instances by more than one user, this creates a multi-user environment in domU. An EC2 instance can be a Linux-based OS such as a Fedora OS [106] or a Red Hat Enterprise Linux or RHEL OS [109]. We call an EC2 that is a RHEL OS an EC2 RHEL. A domU in this chapter is a replication of an EC2 RHEL. We classify the types of domUs as follows.

- **A domU with a single user:** This is when Alice rents a domU, she will then get two user accounts at the same time. The accounts are a root user account and a standard user account, as discussed in Section 5.1.1. Therefore, she is the only user in this system with two accounts. Each account is composed of a user name and password.

- **A domU with multiple users:** This section discusses two types of multiple users domU.

  - **Multiple users per domU with one root:** This paragraph describes multiple users per domU with one root use, when Alice is the root. This description will be referred to again when we discuss how our proposed logging solution is useful for this multi-user environment in domU, in Section 6.2. This is when more than one user shares a domU which can be configured, as described in [110]. Note that all Alice's files that we are considering are in diskU such as in '/home/alice/' directory, which is a virtual disk of Alice's domU. It is virtually in hwU, but physically in disk0/hw0, as discussed in Section 3.3.2.

For example, this can be done after Alice rents a domU with a single user. Thus, she will have both the root account and 'alice' account. Later, she adds Bob as 'bob' account to her domU as a standard user. Thus, this domU can be used by multiple users such as Bob and Alice, and there are three user accounts in Alice's domU. The first one is a root user account. Alice knows this root user account's password. The second account is that of alice user. This account is the first standard user of this domU and is owned by Alice. This account's password can be different from the root user account's password. Thus, Alice owns two accounts which are the root account and the alice account. Then, she knows two passwords which are the root user account's password and the alice user account's password. The third account is bob user which is a standard user. This account is created by Alice for Bob to log in to and share Alice's domU.

A standard user can only access his or her files in specific locations which are managed and provided by the system. Each standard user cannot access other standard users' files. For example, a directory for alice user to store her files can be '/home/alice'. Bob cannot access the '/home/alice/' directory, but a root user could. As discussed in Section 5.1.1 a root user can access all parts or files of the system [105]. Thus, the root user in Alice's domU can access any file in '/home/alice/'. The root user can also execute any executable files such as programs or applications in this directory, but bob user is unable to.

– **Multiple users per domU with more than one root:** For example, after Alice rents a domU she is the root user and administrator. Then she adds a new root user as 'rootusr' using approaches described in [108]. Therefore, rootusr has the same privilege as root which has full access to all files in this domU. Both Alice and the new rootusr can log in to this domU with their own individual and separated root passwords.

This environment can be set up by separating key-pairs [111] for separate root users as described in [108]. Apart from the domU usage scenarios described above, there could be more usage scenarios. For example, each root can add their own new standard or root users to their domU. However, this study is limited to only the scenario described above.

5.2 Customers' Critical Files in DomUs

This section defines customers' critical files, explains the files' location in the generic logging components, and give examples of the files.

This thesis defines **critical files as files in diskUs that are owned by customers or domU's owners. They are the customers' asset and valuable for their businesses**. Thus, the customers do not want anyone to access these files apart from themselves and their authenticated users, and do not want loss or leakage of the files. For instance, [76] states that cloud customers fear loss of control of their financial and health data, which are usually processed remotely in unknown cloud provider's machines. A critical file can be any file such as a text, executable, or database file.

Figure 5.1 shows the location of a critical file (called f) in the diskU within the generic logging components. Critical files can be created inside domUs, or may be uploaded via the Internet by customers from their local machines to diskUs in the cloud. However, whether customers' files will be considered as critical or not is depended on consideration of the customers.

For EC2, customers upload their files to the domU to run their systems. In [18], in order to more quickly find new drugs to heal new diseases, a computer intensive scientific experiment can be conducted in EC2 domUs. After these EC2 domUs are launched, all essential software, the input files, and executable C program files are transferred to the diskUs [18]. Thus, all these files in the diskUs are critical from the point of view of the owner of these domUs.

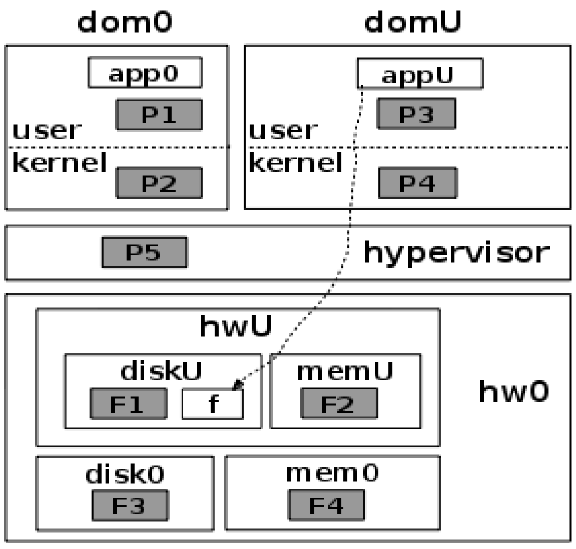


Figure 5.1: From the generic logging components, showing the location of a critical file f in diskU

5.3 Measuring the costs of Threat 1

This section discusses how to measure the costs of threat 1 which affects both the customer and the provider, based on the costs of cybercrime.

Threat 1 as cybercrime. Again, threat 1 is the *abuse and nefarious use of cloud computing*. This could be when criminals rent, own, and then use domUs to conduct their activities [23]. We discuss the effects on both the customer and provider due to threat 1 by focusing on when attackers use domUs to conduct their criminal activities which can be considered as cybercrime.

European Commission [112] argues that cybercrime is understood as "criminal acts committed using electronic communications networks and information systems or against such networks and systems". They state that, in practice, the term cybercrime is applied to three categories of criminal activities. One of the categories, which relates to threat 1 is "crimes unique to electronic networks, i.e. attacks against information systems, denial of service, and hacking". We consider criminal activities conducted by attackers using domUs as 'cybercrime' which can have an effect on both providers and customers. The effect can be measured by direct losses and indirect losses of both the customer and provider side.

Direct losses and indirect losses because of threat 1. [113] defines direct losses and indirect losses as follows. Direct loss is the monetary equivalent of losses, damage, or other suffering felt by the victim as a consequence of a cybercrime. An indirect loss is the monetary equivalent of the losses and opportunity costs imposed on society by the fact that a certain cybercrime is carried out, no matter whether successful or not, and independent of a specific instance of that cybercrime. Indirect costs cannot generally be attributed to individual victims. [114] also states that the indirect costs associated with cybercrime can include such factors as reputational damage to organisations, loss of confidence in cyber transactions by individuals and businesses, reduced public sector revenues, and the expansion of the underground economy.

[113] also states that although the direct costs of cybercrime for the typical citizen are in the tens of pence or cents, the indirect costs are much higher. For example, the botnet[1] behind a third of the spam sent in 2010 earned its owners around US$2.7m, while worldwide expenditures on spam prevention was probably more than a billion dollars. In our context, threat 1 can be considered as cybercrime such as when attackers use domUs to conduct their criminal activities, and the thereat also affect customers or providers in terms of direct costs and indirect costs.

5.4 Effects on the Customer due to Threat 1

This section provides an in-depth discussion of effects on customers due to threat 1. The discussion includes: customer's critical files at risk due to threat 1, domU attacks upon domU, and effects on customers regarding direct losses and indirect losses when customers' critical files are attacked. The purpose of this section is to indicate how serious threat 1 can be for customers.

The CSA report [23] provides examples of this threat such as when criminals use domUs as a base for criminal activities. These activities could be when they

[1]The term botnets is used to define networks of infected end-hosts, called bots, that are under the control of a human operator commonly known as a botmaster [115].

use domUs to attack other domUs and dom0. In IaaS, when a customer rents a domU, he or she may maliciously use this domU (we call this domU a malicious domU) to attack another customer's domU (we call this domU a victim domU).

5.4.1 Customer's critical files at risk due to threat 1

This subsection discusses customer's critical files which are at risk due to threat 1. As discussed in Section 5.2, customers' critical files are the customers' asset and valuable for their businesses, and they do not want anyone to access these files apart from themselves and their authenticated users, and do not want loss or leakage of the files. Examples of the customer critical files can be financial, health, or database data files. Form of attacks designated as threat 1, that we are considering is those where criminals are using domUs to attack customer domUs.

Thus, the most serious situation for a customer can be when a criminal is able to use domUs to attack such as damage, read, modify, or delete critical files in customers' domUs. This attack is vital, as also agreed by [8; 30; 43; 76]. There are no obvious publications in the literature of how a criminal can use domUs to attack critical files in other domUs. However, the next subsection (Section 5.4.2) discusses how it is possible that criminals can use domUs to attack other domUs and can obtain victim's domU data such as a private key for decryption[1] from the victim's binary data in shared memory such as CPU cache.

5.4.2 DomU attacks upon domU

This subsection discuses the issues of the cloud sharing infrastructure. The issues enable criminals to use malicious domUs to attack victim domUs. Then, the criminals can obtain the victim domUs' private data.

[1]Encryption and decryption mechanisms are used to transmit messages privately and secretly over networks [116]. Encryption changes the original message or plaintext into ciphertext. Decryption changes the ciphertext back to plaintext. A private key is used in the decryption process.

5.4.2.1 Sharing cloud infrastructure

This is able to happen because, in IaaS infrastructure, a malicious domU and a victim domU share the same machine and the same hypervisor, as argued by [78]. [78] also states that the main source of security threats in the IaaS platform is domUs because a provider hosts domUs without being aware of the domUs' contents, and without control over them; thus this makes it easy to take over the domUs, then a compromised domU can attack the other domUs.

When a malicious domU and a victim domU reside in the same machine and share a hypervisor, this is called cloud co-residency [117]. DomU attacks upon domU, which is a form of attack within threat 1, can have a serious effect on cloud customers, as discussed below. This is because through these attacks, an attacker, particularly a victim's business competitor, may be able to read private data or compromise the victim domU, as argued by [117].

5.4.2.2 Attacks due to sharing cloud infrastructure

[9] argues that when customers move their computing systems to the cloud, security concerns such as VM-level attacks involving computer and network intrusions or attacks will be made possible or at least easier. Cloud providers deploy hypervisor or VM technology to build up their multi-tenant cloud architectures, which causes potential vulnerabilities and then can cause a threat called 'VM-level attacks' [9].

An example of VM-level attacks is cross-VM *side-channel attacks*. Side-channel attacks are attacks against cryptographic implementations and the targets of these attacks can be primitives, protocols, modules, devices, and systems [118]. [119] demonstrates how cross-VM side-channel attacks can extract private keys. Precisely, this paper assumes that (i) when a malicious domU and a victim domU reside in the same machine and are deployed by the same hypervisor (Xen), and (ii) when the victim domU is decrypting an Elgamal ciphertext[1] using a libgcrypt cryptographic library [120]; then the malicious domU can extract an ElGamal decryption key which is a private key in a shared processor or CPU

[1]The ElGamal encryption system is an asymmetric key encryption algorithm for public-key cryptography.

cache.

Loss of a private key can cause many issues in security. For example, [41] agrees that this key can be used by a web server application to establish secure channels with its clients; if an attacker has the private key of this application, they can impersonate the server before its clients. Thus, if a malicious domU attacks a victim domU and gets the private key of the victim domU's business web server using cross-VM side-channel attacks, this can have a serious effect for the customer who owns the victim domU.

Moreover, [121] also demonstrates that when a malicious domU resides in the same server as a victim domU, it is possible that the malicious domU can monitor how access to resources varies for the victim domU then potentially gather conscious data about the victim. The CSA argues that domUs can host malicious software which has proven especially effective in compromising critical private resources in cloud environments [122]. Although they did not mention to whom the critical private resource is belonging to, it may have belonged to customers.

5.4.3 Effects: Direct losses and indirect losses when customer's critical files are attacked

This subsection discusses the direct losses and indirect losses when customers' critical files are attacked. As stated in Section 5.3, direct loss is the monetary equivalent of losses, damage, or other suffering felt by the victim as a consequence of a cybercrime. Concerning the effects on a customer due to threat 1, this cybercrime can occur when attackers use appUs to damage such as alter costumer's critical files. This can be a vital incident. This incident causes both direct costs and indirect costs for cloud customers. With regards to the direct costs, attackers may ask the victims for money to get the files back. However, the indirect costs for the victim's business are much higher, the same as other cybercrimes that are discussed in Section 5.3. For example, customers may suffer from reputational damage to organisations, loss of confidence in cyber transactions by individuals and businesses, reduced public sector revenues, and the expansion of the underground economy.

5.5 Effects on the Provider due to Threat 1

This section provides an in-depth discussion of the effects on providers due to threat 1. The discussion includes three main aspects. First is that hypervisor or dom0 vulnerabilities may be channels for domUs to attack the provider's cloud infrastructure. Second is that because of these vulnerabilities, it is possible that domUs can attack a dom0 and use this dom0 to attack other domUs. Third is a discussion of direct losses and indirect losses when providers are attacked by domUs.

5.5.1 Hypervisor or dom0 vulnerabilities

This subsection discusses hypervisor or dom0 vulnerabilities. Incidents caused by these vulnerabilities may be channels for domUs to attack the provider's cloud infrastructure.

[123] states that a guest VM such as a domU can infiltrate and hack its host system. [124] argues that Xen as a hypervisor is still vulnerable. In a shared infrastructure environment, a dom0 and a number of domUs reside in the same machine; thus it is difficult to clarify the borders between the dom0 and domUs [125]. Thus, this environment can be vulnerable.

Again, it can be crucial when the dom0 is compromised because then all domUs could be at risk [58]. [126] states that domUs can attack the dom0 that hosts them. [125] agrees that this is because of a difficulty in clarifying the borders between a dom0 and domUs in the same physical machine with virtualisation infrastructure. Thus, these unclear borders can be one of the attack channels. Another channel can be vulnerabilities in the dom0. The vulnerabilities, such as holes in the management consoles of dom0, can allow attackers to gain the root privileged in this dom0, as argued by [59].

After a dom0 is compromised, it can be used by attackers to monitor domUs, eavesdrop on communications between domU and dom0, take control of all domUs, and inject malware into domU images [127]. The main purpose of malware is to damage computers such as alter or delete files, and install other malware [128].

5.5.2 DomU attacks upon dom0 and use of this dom0 to attack other domUs

This subsection discusses in detail, because of the hypervisor or dom0 vulnerabilities discussed in Section 5.5.1 above, how domUs can attack dom0 and use this dom0 to attack other domUs.

Virtualisation vulnerabilities in Windows 2008 as a hypervisor allow the domU running under the hypervisor to crash the Windows 2008 host as dom0 [126]. Thus, a domU can control a dom0 and exploit the other domUs hosted on the same physical machine [129]. After the dom0 is compromised, attackers can get control of the entire domUs [74]. Thus, they may obtain root accounts of these domUs, log into them, and use appUs to access the domUs' critical files, see Figure 5.1, and discussion in Section 5.2 above. If these files are the customers' business databases, this can be a very serious incident.

Moreover, [59] argue that this is because the dom0 can transparently read and write the memory content of the domU using the management interface; thus, if a dom0 is compromised by attackers, they may use the management interface to steal the valuable information from any domU. It is also argued by [46] that a dom0 can access all data in diskUs. This can be a serious security concern from the point of view of the customers.

It is critical when a dom0 is compromised. For example, [41] demonstrates how to use dom0 to obtain confidential data from a domU. The dom0 may control domUs and access critical files in domUs. They may inject malware into these domUs' images (as discussed in Section 5.5.1) to track these domUs' activities. After malware compromises a computer, it can steal data at the OS or file system level [130]. Thus, it may access, lose, or leak (threat 5) critical files, which is a serious incident.

There are no explicit experiments in public literature of domUs compromising dom0s. However, the discussion above can support that this incident may be possible.

5.5.3 Direct losses and indirect losses when providers are attacked by domUs

Subsection 5.5.2 already discussed in detail, because of the hypervisor or dom0 vulnerabilities discussed in Section 5.5.1, how domUs can attack a dom0 and use this dom0 to attack other domUs. This subsection discusses the direct losses and indirect losses, which result when providers are attacked by domUs. Although it seems that the incidents caused by threat 1 discussed above can affect only customer security concerns, this threat can also affect providers.

Direct losses. This thesis argues that threat 1 can also cause the compromising of the dom0. As discussed in Section 5.3, direct loss is the monetary equivalent of losses, damage, or other suffering felt by the victim as a consequence of a cybercrime. Thus, when the dom0 is compromised, the provider's direct loss can be the monetary equivalent of losses, damage, or other suffering felt by the provider as a consequence of the compromising of dom0. For example, the damage could be of the provider's cloud infrastructure which is attacked by the attackers.

Indirect losses. The compromising can allow attackers to control dom0, and use it to compromise all domUs. Then, attackers (especially the competitors of the victim domUs) may access, lose, or leak customers' critical files, which is threat 5.

If the customers find out this fact, it can impact on the providers in terms of indirect losses which is losing their business reputations, the same as the impact on the customers, and as discussed in Section 5.3. The provider companies' reputations can be important for customers when deciding to buy cloud products [33; 131]. Thus, if the customers know that a lot of criminals are or have been inside the provider's cloud infrastructure (threat 1), they may not want to buy the product from this provider.

A high number of malicious domUs in a providers' cloud infrastructure can affect the providers' business reputation. Apart from criminals using domUs to attack domUs and the dom0, as discussed above, there are some other forms of attacks designated by threat 1. The examples of these forms can be domUs that host spamming activities, or downloads for illegal software [23]. If there are a lot of criminals inside a provider's infrastructure, this can impact on the provider's reputation, because this can be an indicator of vulnerabilities in the infrastructure. Thus, this thesis argues that the forms of attacks within threat 1 can affect providers' businesses. Thus, this thesis discusses how to identify spamming activities in a domU as an example of how our proposed solutions can mitigate the risks associated with threat 1 to the benefit of the provider (Section 5.6.4 and 6.3.2).

5.6 History of Critical Files and Process logs

This section provides a definition of the history of a critical file and examples of the history information in Subsection 5.6.1, and a discussion of how the history of critical files can be useful for both customers and providers in Subsection 5.6.2. Then, it discusses a process behaviour log and the log's examples in Subsection 5.6.3, and how the process behaviour logs can be useful to deal with spam activities in domUs in Subsection 5.6.4.

After that, it discusses how the process behaviour logs can be useful to enhance the flexibility of auditing in the cloud using logging systems in Subsection 5.6.5, and how both the history of critical files and process behaviour logs can provide indirect benefits for providers in Subsection 5.6.6.

5.6.1 History of Critical Files to Mitigate the Risks Associated with Threat 1 for Both Sides

This subsection defines the history of a critical file, and provides examples of the history information.

As discussed in the compromising of both domUs or the dom0 in Section 2.5, and Section 5.2 to Section 5.5, either domUs or the dom0 compromising may have the result of undesired access to, or loss or leakage of, customers' critical files. Thus, we propose to maintain a history of each of these critical files. This thesis applies work by PASSXen [43] and HP TrustCloud [27] to form the definition of the history of a critical file. PASSXen is a system that can collect the information on the creation, access, and destruction of a file in the domU. TrustCloud is a framework to deal with the lack of trust in the cloud. It has file-centric information in domUs. This information is obtained by tracing domUs' data and files since they were created until deletion. Typical file operations are read, write, create, and delete [2].

The history of a critical file is the information about the file since it was created, until permanently deleted. Precisely, it contains records of three periods of a critical file's life time:

- created or when the critical file is created,
- accessed or when the critical file is read or writte, and
- destroyed or when the critical file is deleted.

For the contents of the history of any critical file, the main attributes of each record of the contents are as follows:

1. a record number,
2. a process id of the process that is conducting an operation such as read or write on the critical file,
3. a process name of the process that is conducting an operation on the critical file,
4. the id of the owner of this process,
5. a timestamp of the operation, and
6. the operation name including create, read, write, delete.

Table 5.1 is an example of the history of a critical file in a domU. The attributes in the table are listed below:

1. rec_no (row 1, column 1) is a record number,
2. p_id (row 1 column 2) is a process id of the process that is conducting an operation on the critical file,

rec_no	p_id	p_nm	p_ownId	time	operation
1	4001	appU1	1001	t1	create
2	4002	appU2	1001	t2	write
3	4003	appU3	1001	t3	read
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
N-1	4999	appU2	1001	tN-1	write
N	5000	appU3	1001	tN	delete

Table 5.1: The content of the history of a critical file in a domU, N is a number of records

3. p_nm (row 1, column 3) is a process name of the process that is conducting an operation on the critical file,

4. p_ownId (row 1, column 4) is the id of the user that runs this process,

5. time (row 1 column 5) is a timestamp of the operation, and

6. operation (row 1, final column) is the operation name.

The detail of records in the table are listed below.

1. rec_no 1 or the first record is the evidence that a critical file is created in diskU. For instance, the content in the first record, row 2, or rec_no 1 is the evidence or log that a critical file is created by a user where his or her id is 1001 in rec_no 1 column 4.

2. rec_no 2 to rec_no N-1 or the records between the first and the last record is evidence of when the critical file is accessed such as read or write. For example, rec_no 2 is evidence or log content of when appU2 (rec_no 2, column 3) writes (indicated by rec_no 2, the last column) to this critical file.

3. rec_no N or the last record is the evidence of when this critical file is permanently deleted from diskU. For example, the last record or rec_no N is the evidence when this critical file is permanently deleted from diskU by

a process with its name 'appU3' in rec_no N, column 3. AppU3 is executed by a user with the id 1001 in rec_no N, column 4.

rec_no	p_id	p_nm	p_ownId
1	4624	read	1002
2	4800	read	**1003**

Table 5.2: The content of the history of critical file f or s.txt, when the appU or process called 'read' is reading the file.

An example of the content of the history of critical file s.txt. If a critical file f in Figure 5.1 is s.txt, the content of history of s.txt as a log file can be for instance seen in Table 5.2. To simplify further discussion, for this table, we assume that appU or process called 'read' is reading s.txt. Thus, we know an operation name which is 'read', and we know the critical file name which is s.txt. In the table, a timestamp of the operation or column 5 in Table 5.1 is omitted and will be discussed in Section 5.7.5. The operation name or the last column in Table 5.1 is also omitted. This is because we already know the operation name. Table 5.2 presents only two records of the history of a critical file s.txt. The details of both records are listed below. We omit the file creation and deletion record. It is assumed that 1002 is Alice's id and 1003 is Bob's id.

1. For rec_no 1 in this table, p_id as 4624 and p_nm as read is the id and name of the process that reads this file respectively, and p_ownId as 1002 or Alice's id is the id of the user who runs this process.

2. For rec_no 2 in this table, 4800 and 'read' is the id and name of the process that reads s.txt respectively, and 1003 as Bob's id is the id of the user who runs this process.

Main discussions about how to obtain all this information from Table 5.2 are in Section 5.7 and Section 6.1. The contents of the table can be more complex to provide more precise evidence, which will be discussed in Section 6.2 and 6.3.

It is possible that some incidents happening in domUs can have negative effects on customers or providers. For example, the negative effects on customers can be when s.txt is belonging to Alice, from rec_no 2 in Table 5.2, why and how can Bob access Alice's critical file. The history of critical files or rec_no 2 in Table 5.2 can be used as evidence to assist in clarification of what happened with Alice's domU. This kind of evidence such as Table 5.2 can be a clue to discover what is going on inside domUs that contain critical files. Section 5.6.2.1, 5.6.2.2, 5.6.3, 6.2, and 6.3 discuss how to discover the causes of the incidents. Consequently, the evidence should assist in mitigating the risks associated with threat 1 such as criminal domUs. As a result, this should assist in mitigating the causes of negative impacts on both customer and provider companies such as brand damage, as discussed in Section 5.3, 5.4, and 5.5.

5.6.2 The history of critical files, work for both sides

This subsection discusses in detail how the history of critical files can be useful for both customers and providers.

[43], [30], and [76] argue that a file-centric log is important when dealing with the cloud problems. The history of critical files (Section 5.6.1) is also a file-centric log. To discuss how the history of critical files works, we assume that an attacker can log into a domU and conduct criminal activities in this domU. For example, the attacker may use appUs in this victim domU to access the domU's critical files (as will be discussed in Section 5.6.2.1) or to send spam emails as will be discussed in Section 5.6.2.2. In this situation, the history of critical files (Section 5.6.1) which is produced by logging systems can assist in auditing or clarifying the one who is responsible for this situation.

5.6.2.1 The history of critical files to mitigate the risks associated with threat for the benefit of the customers

This section discusses how the history of critical files can assist in mitigating the risks associated with threats, to benefit the customers.

It would be useful if we had a history of each of the domU's critical files to assist in indicating, for example, who has access to these files, and which appU have accessed them. The history information can enhance accountability in the cloud and customers' confidence. For example for threat 1, when Alice's domU is compromised by attackers, they may control appU to access her critical file f or s.txt, as shown in Figure 5.1 and discussed in Section 5.4. The history of f can be used to clarify this undesired malicious incident by the attackers. For example, the history can be information about which appU accessed f, when, and whom this appU belonged to.

The history of critical files could assist in analysing attacker behaviours inside domUs. To analyse attacker behaviours, Alice can check rec_no 2 column 4 in Table 5.2, and may discover that someone else such as Bob accessed her critical file s.txt. The content of the table can be more complex to provide more precise evidence, which will be discussed in Section 6.2.

5.6.2.2 How the history of critical files helps providers to deal with dom0 compromising

This section discusses how the history of critical files helps providers to deal with dom0 compromising.

In Table 5.2, when one discovers that Bob has accessed Alice's critical file s.txt, this can be a trigger for the providers to be aware that their dom0 may be compromised. To identify Bob accurately, they can conduct further investigation, for example, by pinpointing Bob's Id (p_ownId, column 4), the appU name (p_nm, column 1) he used to access s.txt, or the s.txt file name. Then, they can gather more necessary evidence. For example, this can be achieved by recording Bob's appU behaviours as done in the case study in Section 3.4.2, or monitoring this appU or domU for malicious network traffic as done in [1]. Thus, the history of critical files can be of benefit to the provider.

5.6.3 Process Behaviour Logs

This subsection discuses a process in domU, defines a process behaviour log, and provides examples of a process behaviour log.

5.6.3.1 Processes in DomUs

A process is a program in execution [132], or the operating system's abstraction for a running program [133]. An appU can, for example, be a C programming code, and is stored in the domU/diskU. After a user compiles this code, she can obtain the appU application/program. Then, in Linux system, she can run this application by the command './appU'. The CPU executes the instructions in the program, this is when the appU application become a process called 'appU'. This thesis uses Linux processes as an example of processes in a domU. Actually, in the generic logging components (Figure 3.1), an appU is composed of one or more processes or commands. An appU life cycle is when the appU is executed and becomes a process, until the process is terminated. For each individual appU life cycle, the process has a unique id.

5.6.3.2 A processes behaviour log definition

Log files or a provenance collection in PASSXen [43] system include the creation, access, and destruction of the processes of domUs. **A process behaviour log is a record of the process since it was created until it is destroyed**. Precisely, it contains records of three periods of a process life time: the creation, access, and destruction of processes in the domUs.

Details of the periods are discussed in the lists below.

1. For a period of process creation, a user level process can be created by executing an executable program or application [134]. We define a period of process created as when an appU is executed. This causes a live process named 'appU' which exists in memU.

2. A period of process access is when a process has access to a file. This period focuses on the file operation (read or write) that a process carries out on this file.

3. To define a period of process destruction, [134] states that a user process destruction can be driven by a normal process termination event. We define a period of process destruction as when an appU is normally terminated when all its instructions are executed. This causes a live process name 'appU' to disappear from the memU.

Customers do not want unauthorized users to access their critical files such as financial and health files or business database files, as discussed in Section 5.2. Thus, in each record of the process behaviour log, we focus on what file operation (read and write) a process carries out on a file.

Then, for the contents of the process behaviour log, the main attributes of each record are as listed below:

1. a record number,
2. a process id of the process that is conducting an operation (read operation or write operation) on a file,
3. a process name of the process that is conducting the operation on a file,
4. the id of the owner of this process,
5. a timestamp of the operation,
6. a file name of the file that this process is conducting the operation on, and
7. the operation name including read or write.

rec_no	p_id	p_nm	p_ownId	time	a_file	operation on a file
1	4001	appU	1001	t1	-	-
2	4001	appU	1001	t2	a.txt	write
3	4001	appU	1001	t3	b.txt	read
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
N-1	4001	appU	1001	tN-1	c.txt	write
N	4001	appU	1001	tN	-	-

Table 5.3: The content of process behaviour logs of a process (for an appU life cycle), N is a number of records

Table 5.3 is an example of the process behaviour logs of an appU process, or appU life cycle. In the table, the attributes of each record are listed below.

1. rec_no (row 1, column 1) is a record number,

2. p_id (row 1, column 2) is a process name of the process that is conducting an operation on the file,
3. p_nm (row 1, column 3) is a process name of the process that is conducting an operation on the critical file,
4. p_ownId (row 1, column 4) is the id of the owner of this process,
5. time is a timestamp of the operation (row 1, column 5),
6. a_file is the file name of a file that this process is conducting an operation on (row 1, column 6), and
7. operation is the operation name (row 1, final column).

The details of each record are listed below.

1. rec_no 1, the content in the first record is the evidence that a process is created. For example, the content in the first record is the evidence or log that a process (with its id as 4001 or rec_no 1, column 2 and its name as appU or rec_no 1, column 3) is created by a user whose id is 1001 as in rec_no 1, column 4.
2. rec_no 2 to rec_no N-1, the records between the first and the last record is evidence of when a process has conducted an operation such as read or write on a file. For example, rec_no 2 is the log content when this process writes (rec_no 2, column 7) to a file (named a.txt as in rec_no 2, column 6).
3. rec_no N, the last record is the evidence of when this process is destructed. For example, the last record is the evidence of when this process or appU is permanently destructed from domU by a user id 1001 as in rec_no N, column 4.

5.6.3.3 An example of process behaviour logs

Table 5.4 is an example of a cat process behaviour log file. The command *cat addr.txt* is when a cat command in Linux accesses the text file called addr.txt. *Cat*

p_nm	p_id	a_file	p_ownId
cat	4000	addr.txt	1002

Table 5.4: A process behaviour log file to show the malicious cat command reading addr.txt

stands for concatenate. The content of the log of the cat command can be listed as below:

1. the name of the process (p_nm, cat),
2. the id of the process (p_id, 4000),
3. the name of the file accessed by the cat process, (a_file, addr.txt), and
4. the id of the owner of this process (p_ownId, 1002).

This log can be different, depending on who (a provider, customer, or auditor) wants it and what it is for. The table shows only the content for the purpose of identifying a spam domU in Section 5.6.4.

5.6.4 Process behaviour logs for spam activities

To benefit the providers, this subsection discusses how the process behaviour logs can be useful to deal with spam activities in domUs. To explain how process behaviour logs can assist in mitigating the risks associated with spam activities, an example of three systematic steps to identify a spam domU is presented below.

1. A provider needs to pinpoint a suspicious domU, and a suspicious command or process inside this domU.
2. The provider uses the suspicious domU id or name and the suspicious process name as a trigger for a logging system to capture the suspicious process activities in the process behaviour log for appropriate period of times.
3. The provider or an auditor can audit the log to analyse and identify this suspicious spam domU.

We discuss each step in detail in the lists below.

- For the first step, to pinpoint a suspicious domU and process this for spamming, we assume that a provider has already pinpointed a domU and process/command (e.g., a mail command in Linux system) that is suspected of sending spam emails.

- For the second step, the provider can capture activities of this suspicious mail command as a log file (called Log 1) to be used in the analysis and identification of this spam domU. The details of this step are discussed below.

  - **To capture the mail command behaviour/activities.**

    The case study in Section 3.4.2 already demonstrated how to capture a mail command behaviour, as a simulation of spam activities. In the case study, to send a spam email to a victim, the mail command in the simulation is in the form of *mail -s spamSubj winai.wongthai@ncl.ac.uk*. This command sends the string 'spamSubj' as an e-mail subject to the email address 'winai.wongthai@ncl.ac.uk'. A logger in this case study can capture both the e-mail subject or the string 'spamSubj' and the email address or the string 'winai.wongthai@ncl.ac.uk', and is able to record the captured information as a log file.

  - **To capture another command that works with the mail command.**

    We can take one step further from the case study in Section 3.4.2. One of the forms of this mail command to send emails to more than one email address at once is *mail -s spamSubj addr1 add2 ... addrN*. When addr1 to addrN is stored in a file (e.g., addr.txt), the form of this command can be *mail -s spamSubj $(cat addr.txt)*. The cat command will access addr.txt, then display all the addresses to the mail command. The mail command then sends a spam email to all the addresses. In this case, this form of mail command (*mail -s spamSubj $(cat addr.txt)*) has three arguments. The first argument is '-s', the second is 'spamSubj', and the last is '$(cat addr.txt)'.

    The second argument or 'spamSubj' and the third argument or '$(cat addr.txt)' of the mail command can be evidence to assist in analysing and identifying this mail command as spam activity, and to further identify the domU that uses this command. As we can see, the third argument of the mail command is '$(cat addr.txt)'. The provider can record the cat process behaviour as process behaviour logs (called Log 2 or Table 5.4). The logger in the case study can also capture and record the second argument as 'spamSubj' and the third argument as '$(cat addr.txt)' of the mail command discussed above.

- For the third step, the provider or an auditor can audit the log to analyse and identify this suspicious spam domU. For the command *mail -s spamSubj $(cat addr.txt)*, this mail command sends emails to all victim email addresses in a text file *addr.txt*. Thus, this mail command involves *addr.txt* in Table 5.4, column 3. Hence, this file could be very important evidence to identify these spam activities. Section 6.3.2 discusses an example of the complete process behaviour log file to show the malicious mail and cat command involved in spam activities.

  Regarding *addr.txt*, when the providers have already pinpointed the mail command that sends emails, as demonstrated in Section 3.4.2, they can then combine the capture of the mail command's malicious behaviours (Log1/in Section 3.4.2) with the cat command behaviour log file (Log2/in Table 5.4, this command reading addr.txt) as evidence to assist in identifying this spam domU. Note that, for this discussion, we assume that the second argument as 'spamSubj' and the third argument as '$(cat addr.txt)' of the mail command above is sufficient to identify these spam activities.

  Thus, process behaviour logs, for example cat's behaviour log or Table 5.4, can be useful to assist when analysing and identifying spam domUs in the providers' IaaS cloud. We do not discuss identifying an absolute spam domU. This involves more research on the identification of spam application behaviour.

5.6.5 How Process behaviour logs benefit customers

To benefit the customers, this subsection discusses how the process behaviour logs can be useful to enhance the flexibility of auditing in the cloud using logging systems.

rec_no	p_nm	a_file	p_ownId
1	gedit	a.txt	1002
2	gedit	*s.txt	1002

Table 5.5: A list of all files accessed by gedit appU

To enhance the flexibility of auditing in the cloud using logging systems, for example, Alice may want to know whether a particular appU (such as one that has an ability to view text files e.g., gedit in Linux) has accessed her critical files or not. Thus, process behaviour logs of a particular process could be useful for Alice. For example, the process behaviour logs of gedit could be as in Table 5.5. The table shows a list of all files that were accessed by gedit. Alice could therefore check whether this process reads her critical file or s.txt or not. In the table, she can see that gedit reads s.txt, see in rec_no 2, column 3 in Table 5.5. She may then further investigate this incident. Thus, this type of log file can be useful to enable the flexibility of auditing.

5.6.6 Both history of critical files and process behaviour logs with indirect benefits to providers

This subsection discusses how both the history of critical files and process behaviour logs can present indirect benefits for providers. [8] argues that accountability proactively detects and diagnoses problems, and providers can more easily handle customer complaints. Thus, when both the history of critical files and process behaviour logs are reliable and trusted by an auditor, as will be fully discussed in the beginning of Section 6.3, providers can clarify the false claims generated by customers. This benefits the providers as discussed in Section 5.5, in that customers will buy the cloud based on providers' reputations [131]. They will not buy IaaS products if they know that many criminals reside inside the provider cloud infrastructure. The criminals are in the cloud infrastructure can be one of the forms of attacks enclosed in threat 1.

5.7 Available Approaches to Obtain the History of a critical File

There are two approaches to obtain the history of critical files. The first is **domU memory introspection approach**, for short-memory introspection which will be discussed in this section. The second one is **domU system calls interception approach**, for short-interception which will be discussed in Section 5.8. We will describe both of these based on the generic logging components.

As discussed in Section 5.6 the history of a critical file is the records of three periods of a critical file's life time: the time at which the file is created, accessed, and destroyed, as presented in Table 5.1. This section contains subsections to discuss the memory introspection. Subsection 5.7.1 describes the memory introspection approaches to obtain the history of critical files. Subsection 5.7.2 discusses how the memory introspection can obtain the history of critical files. Then, from Linux kernel's data structures,

Subsection 5.7.3, Subsection 5.7.4, and Subsection 5.7.5 respectively explain how to obtain the full path name of a critical file, the operation's name of an operation that is performed on a critical file, and the last accessed and modified times of a critical file. After that, Subsection 5.7.6 describes how to obtain the creation and deletion records of a critical file. Finally, Subsection 5.7.7 discusses how to obtain process behaviour logs through the introspection approach. This includes the method to obtain process behaviour logs when a process carries out an operation on a file, and how to obtain the process creation and destruction record.

5.7.1 Memory Introspection approaches

This subsection describes memory introspection approaches to obtain the history of critical files.

This subsection only discusses the method to obtain a periods accessed information by memory introspection approach. This is because the periods accessed represents the majority of the contents for the history of a critical file, compared to the periods created (rec_no 1 in Table 5.1) and destroyed (rec_no N in Table 5.1), which should be separate records of each history of each critical file. Section 5.7.6 later discusses how to obtain the periods created and destroyed. The terms memory introspection and introspection are used interchangeably.

Memory introspection is the process of accessing the memory state in the main memory (Random Access Memory or RAM) of a target virtual machine such as a domU from another such as a dom0 [39]. It is one of the virtual machine monitoring approaches. The memory introspection has many applications in security and systems management.

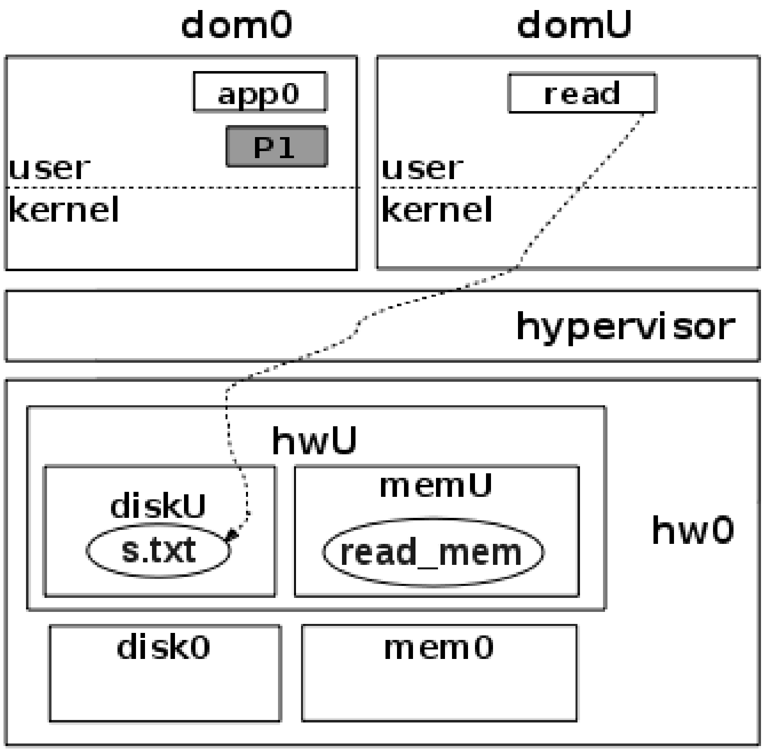


Figure 5.2: Memory Introspection

To obtain the history of critical file, Figure 5.2 illustrates memory introspection in an IaaS environment. From the figure, we can see that when appU (called read) reads s.txt or the dot-arrow line in the figure, this appU will own some memory space in memU, see *read_mem* or the ellipse in memU in Figure 5.2. This memory space includes information of all files opened by this read appU. We can record this information as history of critical files for example by using app0 and P1 (Figure 5.2). This will be discussed later in the following subsections.

The memory space holds all the information we need to record. This information is rec_no 2 to N-1 in Table 5.1 and all records in Table 5.2. However, we only discuss to obtain the history information as shown in rec_no 1 of Table 5.2 as an example how. This information is listed below:

1. a file name of a file s.txt or the string "s.txt",
2. a process Id as 4624 of appU that accesses this file,
3. a process name as read of appU that accesses this file, and
4. an owner Id of the read process as 1002 which is Alice's user Id in this domU.

5.7.2 How the memory introspection can obtain the history of critical files

This subsection discusses how the memory introspection can obtain the history of critical files. This includes: the introspection system architecture, how the logger captures information of the history of a critical file, and Linux kernel data structures for virtual memory organisation of an appU that hold the information of the history of critical files.

This thesis does not propose a new memory introspection tool or approach, as we reuse existing tools. Section 5.6.1 discusses the contents of the history of a critical file s.txt, in Table 5.2.

For rec_no 1 in this table:

1. p_id as 4624 is the id and name of the process that reads this file,
2. p_nm as read is the name of the process that reads this file, and

3. p_ownId as 1002 (Alice's Id) is the id of the user who runs this process.

Section 5.7.2.1 will fully discuss the technical detail of a memory introspection approach which can obtain the information in Table 5.2, by using memory introspection tools and the comprehension of the knowledge of Linux kernel data structures for virtual memory [135].

5.7.2.1 Introspection system architecture

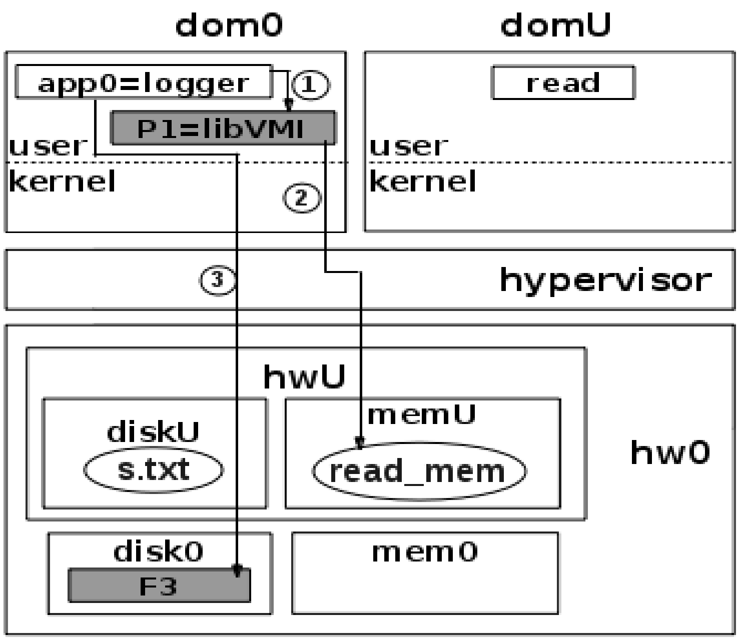


Figure 5.3: The system architecture of the introspection approach

Figure 5.3 illustrates the system architecture of an introspection system. The architecture is based on the generic logging components see Figure 3.1. The main components of the architecture are:

1. app0 or logger in the white box in dom0 user level, Figure 5.3,
2. P1 or libVMI [79] in the shaded box in dom0 user level, Figure 5.3, and
3. F3 as a log file in disk0, Figure 5.3.

LibVMI is a memory introspection tool to read memory from VMs or domUs. Figure 5.3 illustrates the introspection with the approach used in libVMI [79] to obtain runtime access to the memU of a domU. The introspection externally performs, typically from within a dom0. We discuss how to use memory introspection tool such as libVMI to obtain information of the history of critical file s.txt.

We can deploy libVMI in the dom0 user level to read the memory space, see *read_mem* or the ellipse in memU in Figure 5.3. This memory space holds all information (as discussed above) that we need to record, which is a file name of s.txt and rec_no 1 of Table 5.2. Three steps to obtain the contents of the history of critical file s.txt using introspection are listed below. The steps are the circles with numbers in Figure 5.3.

1. The logger in dom0 user level, which is an app0 calls libVMI or P1 to access memU to get the information in *read_mem* such as a file name of s.txt, as discussed above.

2. LibVMI accesses memU to obtain the information in *read_mem* and returns the obtained information to the logger.

3. The logger manages the obtained information then writes the information which is rec_no 1 of Table 5.2 into F3.

Conditions of this system architecture of the introspection approach or Figure 5.3 are listed below.

- How does the logger know whether a file in a domU is critical or not.

  In this prototype architecture, this thesis does not discuss how to manage the logger application. Thus, we assume that the logger already knows a file name of the critical file.

- Why are the logging processes placed in dom0, not domU.

  LibVMI is designed to be inside a dom0. Thus, the logger is also in dom0, to call libVMI. Placing libVMI and the logger in dom0 is better than placing them in domU. Placing both of them in domU allows the owner of the domU to tamper with them, as agreed by [30; 39; 78; 96; 97]. [97] state that when a monitored machine resides in the same machine as the detecting or monitoring system, the owners of the monitoring machine can disable the detection system, or modify the detected data, according to their needs. Moreover, [39] agree that to obey typical good programming guidelines, or good security guidelines for designing monitoring architecture, a monitored machine should not tamper with (e.g., alter the code of) the monitors. Thus, we have all logging processes in dom0, not domU.

5.7.2.2 How the logger captures information of the history of a critical file

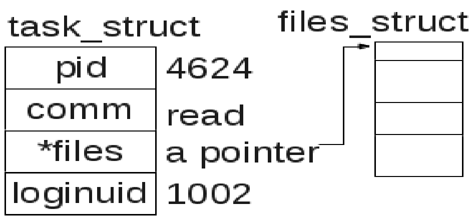


Figure 5.4: A *task_ struct* structure of the read process, adapted from [2]

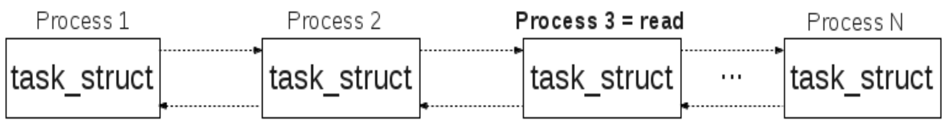


Figure 5.5: task_struct structure list, adapted from[3]

This section discusses the technical details of how the logger (the white box inside dom0 user level in Figure 5.3) in the introspection system architecture works with Linux kernel data structures for virtual memory organisation to capture the contents of the history of critical file s.txt. Linux kernel data structures are built-in data structures for use in Linux kernel code; and virtual memory allows a process to allocate and manage main memory as if it is only one process in the memory of the system [2].

5.7.2.3 Linux kernel data structures for virtual memory organisation of an appU

This section explains how Linux manages virtual memory for its processes, as numbered below.

1. **Linux kernel data structures.**

   The Linux system provide kernel data structures for virtual memory organisation. Each process has its own data structure in the memory. This data structure is task_struct as shown in Figure 5.4. All task_struct structures are linked as circular linked lists as shown in Figure 5.5. In the figure, process 3 represents the read appU.

   Each *task_struct* contains numerous variables to keep track of a process in the memory when it is executed by the CPU [2; 132; 136]. Examples of the variables are: pid or process id; and comm or process name. They hold information which is used to keep track of a process in the memory when it is executed by the CPU. We can use an introspection approach to extract information from all variables in a *task_struct* structure as shown in Figure 5.4. In the figure, an asterisk (*) in front of a variable indicates that this variable is a pointer. These variables are:

   - pid or the process Id of a process such as 4624,
   - comm or a process name or executable name such as 'read',
   - files or a pointer to *files_struct* structure that holds all files that are opened by this process, and
   - loginuid or an account, that includes a user id and that a user uses to gain access to the system [136] such as 1002.

   These variables are based on a 64-bit Linux kernel version 3.8.5 for Fedora 18. The source code repositories of this kernel version can be found at [137].

2. **A Linux kernel data structure for virtual memory organisation of the read appU/process.**

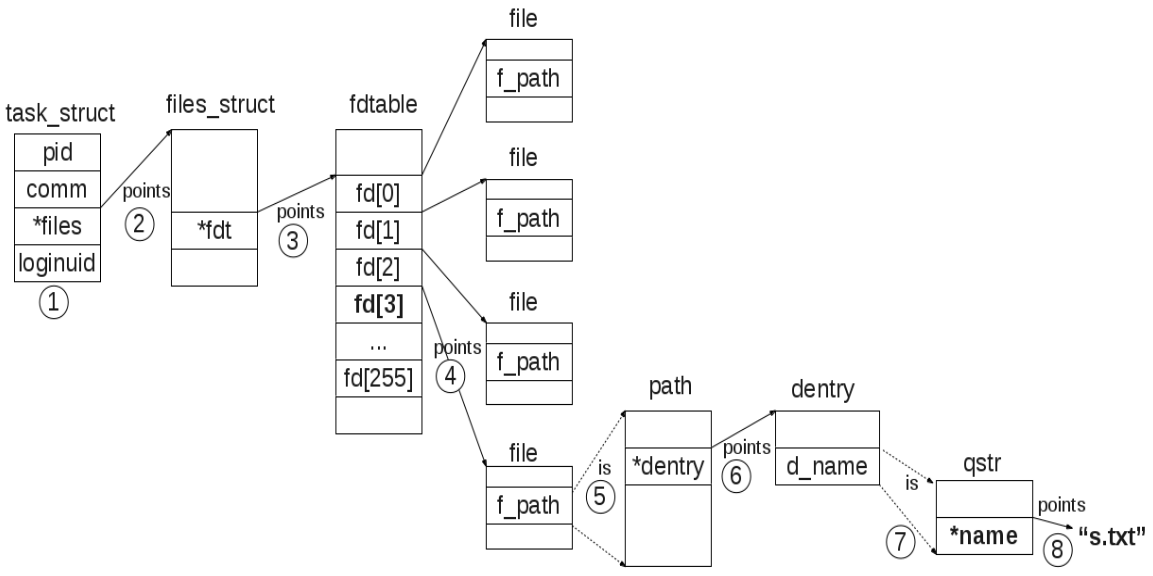


Figure 5.6: Linux kernel data structures for virtual memory organisation of appU process (called read) that reads s.txt.

Figure 5.6 is a Linux kernel data structure for virtual memory organisation of an appU (called read). We assume that this memory organisation is for when a process or appU called 'read' is reading a file called 's.txt'. Thus, this memory organisation exists in the main memory of a Linux system when only the system is running this process and the process is reading the file.

As discussed in Section 5.7.2.1, this memory organisation is a memory space, see *read_ mem* or the ellipse in memU in Figure 5.3. This memory space is owned by read appU when this appU becomes a live process in memU and when it reads s.txt.

Figure 5.6 does not include all the information of the kernel data structures. However, the figure assists in the understanding of our discussion. The logger code traverses all those structures until it gets the history of a critical file information such as the name of s.txt. In Figure 5.6, each box is a whole data structure or data type such as *task_ struct*.

The name above each box is the name of a data structure such as *task_ struct*.

The name inside a box or data structure is a member or variable (e.g., *files* in *task_struct*) of that structure. An asterisk (*) in front of a variable indicates that this variable is a pointer. Each arrow line corresponds to memory address pointers for example the arrow line from variable *files* in *task_struct*. Each pair of arrow-dot lines from a variable to another box (e.g., the pair of arrow-dot lines from f_path to *path*) indicates that the box is a data structure of this variable.

The Linux kernel manages the task data structure or *task_struct* for a process. A *task_struct* structure is responsible for storing all essential information to run a process including process id and pointers to all files that are opened by a process. For example, 'files' is one of the members of a *task_struct*, see Figure 5.6 in the task_struct box. It is a pointer to a *files_struct* data structure which maintains all current files that this process is currently using.

One of the current files is s.txt which the logger needs to capture the file's details such as the file name. When assuming that appU is reading s.txt, the logger can traverse the virtual memory space of read appU in Figure 5.6 by following Step 1 to 8 to obtain the contents of the history of critical file s.txt, as discussed below. Each step is a circle with a number in Figure 5.6.

Figure 5.6 shows Linux kernel data structures for virtual memory organisation of appU process (called read) that is reading s.txt. The lists below present explanations of the data structures involved for the reading event.

- *task_struct*: Figure 5.6 shows that an appU process represented by the *task_struct* structure (the first box from the left in the figure) has the 'files' filed which is a pointer to the *files_struct* data structure which is the second box from the left in the figure.
- *files_struct*: in the figure, a *files_struct* structure contains an fdt pointer, see inside the *files_struct* structure box in the figure. This structure holds information about all of the files that this process is currently using.
- fdt: an fdt stands for '**f**ile **d**escriptor **t**able'. When a process opens a

file, the opened file is represented by a 'file descriptor' which contains the details of this open file [138]. Thus, a file descriptor table or fdt stores all file descriptor of all files. An fdt points to the first open file of a process. This first open file is fd[0] which is a member in the *fdtable* data structure, see inside the third box from the left in the figure.

- *fdtable*: each running process has a file descriptor table which is represented by a *fdtable* structure. This structure contains pointers to all open files and is stored by the kernel [139]. The fdt pointer (see inside the *files_struct* structure box) points to the first open file or fd[0] inside *fdtable* structure. This structure contains pointers for up to 256 *file* data structures. These pointers are fd[0]-fd[255].
- fd[i]: fd stands for file descriptor. Each fd[i] points to each open file that is opened by the read process. Each open file is structured as a *file* structure, see the forth box from the left in Figure 5.6.
- *file*: a *file* data structure is a representation of each open file [135]. This structure contains a variable called f_path.
- f_path: f_path contains a path file of an open file. A data structure of f_path is *path*, see the third box from the right in Figure 5.6.
- *path*: a *path* data structure contains a dentry pointer. This pointer leads us to *dentry* structure which contains a variable called d_name. *Dentry* structure is the second box from the right in Figure 5.6.
- *dentry*: a *dentry* stands for a '**d**irectory **entry**' which is each component of a path [2]. For example, when a path is '/home/alice/s.txt', this path is composed of four *dentry* structures including: the root directory '/', two directories including home and alice, and the file s.txt. Each *dentry* structure contains a variable called d_name which is a *qstr* structure or the last box from the left in Figure 5.6.
- *qstr*: a *qstr* structure has a pointer variable called name. This variable points to a file name of an open file. In this case when a process reads s.txt, it points to a string "s.txt".

3. **The logger obtaining the contents of history critical file s.txt.**

Figure 5.2 shows a read appU reading s.txt, the dot-arrow line and Figure 5.3 shows us the logger (the white box in dom0 user level) needs to obtain history information of the critical file s.txt. Again, the history information are the lists below:

- rec_no 1 in Table 5.2;
  - process id or p_id of a process (called read) that is reading s.txt,
  - process name p_nm of a process (called read) that is reading s.txt,
  - an id of a user who runs the read process, and
- a file name of s.txt.

The logger can traverse the virtual memory space of read appU in Figure 5.6 by following Step 1 to 8 listed below to obtain the contents of the history of critical file s.txt. Step 1 is to obtain the process id and name, and the id of the user who runs read process. Step 2 to Step 8 is to obtain the file name of s.txt.

- Step 1: This step captures the process id and name of the read process that is reading s.txt, and the id of the user who runs the read process, see Figure 5.6. In this step, the logger needs to access the list (Figure 5.5) of *task_struct* structures to capture a read process's *task_struct*, see the third element from the left in the list in Figure 5.5. Then, the logger can capture details of the read process including: the process id and name of the read process, and the id of a user who runs the read process, see Figure 5.6.
- Step 2: In Figure 5.6, a files pointer is the third element from the top of the first box from the left. This pointer is in read appU process's *task_struct*. It points to *files_struct* structure which is the second box from the left in Figure 5.6. This structure contains an fdt pointer.
- Step 3: The fdt pointer is the element of *files_struct*. It points to the first open file or fd[0] inside the *fdtable* structure which is the third box from the left in Figure 5.6. *Fdtable* structure contains pointers

fd[0]-fd[255]. Each fd[i] points to each open file that is opened by the read process.

- Step 4: It is assumed that the read appU is reading s.txt. Thus, s.txt is structured as a *file* data structure. This file is pointed to by a fd[3] pointer.
- Step 5: In Step 4, *File* data structure of s.txt is pointed by fd[3]. Then, in Step 5, the f_path variable (inside *file* structure) is a *path* structure which is the third box from the right of Figure 5.6. This structure contains a dentry pointer.
- Step 6: A dentry pointer is the element in the third box from the right of Figure 5.6. It points to the *dentry* structure which is the second box from the right of Figure 5.6. This structure contains a variable called d_name.
- Step 7: D_name is the element in the second box from the right of Figure 5.6. This element is a *qstr* structure which is the last box from the left in Figure 5.6. This structure has a pointer variable called name that points to a file name of s.txt or a string "s.txt".
- Step 8: The logger can obtain the string file name of s.txt.

Memory introspection, along with Linux kernel data structures for virtual memory organisation of a process, enables logging systems to record the appU operations (e.g., read or write) on a file which is in diskU. This recorded information should be useful for building the logs or history of each file. As a result these logs can be used as evidence to indicate who is responsible for any malicious activity in domU. This can assist in mitigating the risks associated with the seven CSA threats.

5.7.3 Getting the full path name of the critical File

This subsection explains how to obtain a full path name of the critical file.

In the system architecture of the introspection approach or Figure 5.3, s.txt which is in diskU in the figure can actually be in a directory. When we assume that the file is in a directory '/home/alice', thus the full path name of this file is '/home/alice/s.txt'. This section discusses how to obtain the full path name.

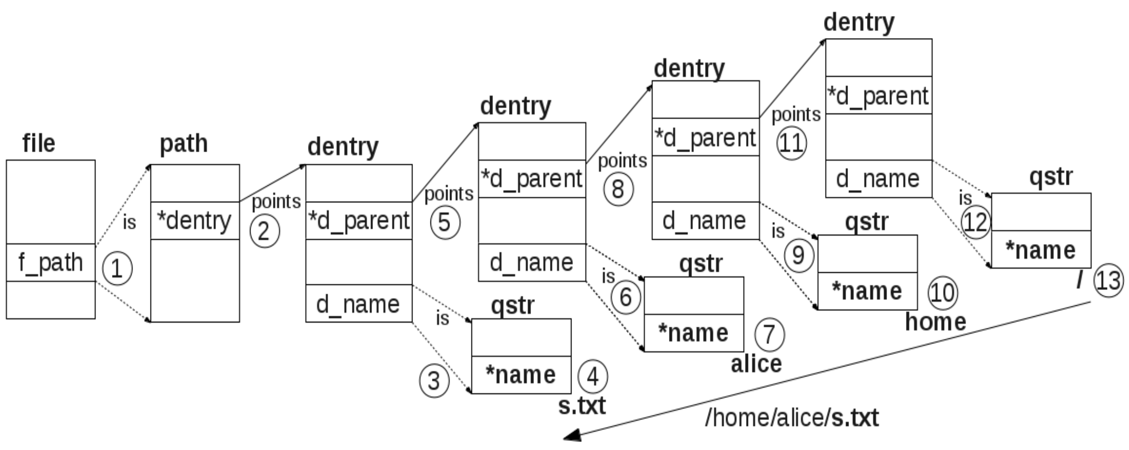


Figure 5.7: How to obtain the full path file name of s.txt

In Section 5.7.2.3, after the logger gets the file name of s.txt (Step 2 to 8, in Figure 5.6), Figure 5.7 shows how the logger get the full path name of s.txt. It is assumed that the logger was already able to get the file name 's.txt' by follwing Step 1 to 4 in Figure 5.7 or Step 2 to 8 in Figure 5.6. To be simplified for explanation, we will follow Figure 5.7 for this. In the figure, the full path name can be obtained after Step 4.

- In Step 5, the logger can follow a d_parent pointer which is the element in the third box from the left in Figure 5.7. This pointer then recursively points to *dentry* structures, see Step 8 and 11. These *dentry* structures are the second, third, and forth *dentry* boxes from the right in Figure 5.7. They hold a parent directory name of s.txt, which is '/home/alice/'. This can be obtained by following Step 6 to 13.
- Then, in Steps 6 to 7, the logger can obtain the 'alice' directory name.
- In Steps 8 to 10, the logger can continue to follow the d_parent pointer, in Step 8. In Step 9 and 10, then the logger can get the parent directory name of the alice directory which is 'home'.

- Finally, in Steps 11 to 13, the logger will reach and get the root directory or '/', and achieves the full path of s.txt or '/home/alice/s.txt', considering all 13 steps).

5.7.4 Getting the Operations of a Critical File

This subsection explains how to get the operation's name of an operation that is performed on a critical file from Linux kernel's data structures. An operation name is already discussed in Subsection 5.6.1, and shown in row 1, the last column of Table 5.1.

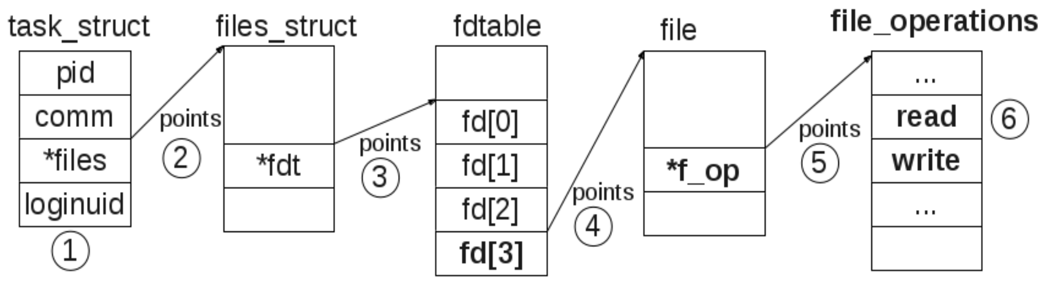


Figure 5.8: How to Obtain File Operations

File_ operations is a data structure that has pointers to functions such as read() or write() [135; 140]. Each field of the structure corresponds to the address of the functions. A process can call these functions when it is performed on an open file [2]. A pointer member of the *file* structure called f_op points at *file_ operations* [135].

Figure 5.8 shows that how a logger can ascertain the functions that a process invokes on an open file, outlined in the lists below.

- The routines of Step 1 to 4 in the figure is the same as those in Figure 5.6.
- In Step 5, the logger can locate the f_op pointer or the element in the forth box from the left of Figure 5.8. The logger then follows this pointer to the *file_ operations* structure or the last box from the left of Figure 5.8.
- Finally, the logger can get a function (e.g., read or write) in Step 6.

5.7.5 Getting the last accessed and modified times of a file

This subsection explains how to get the last accessed and modified times of a critical file from Linux kernel's data structures.

Unix or Linux systems separate the concept of a file from any correlated information about this file [2]. For example, the correlated information about a file are access permissions, size, and owner. This information is stored in a separate data structure (called an inode) from the file. Thus, an inode is a data structure that keeps track of all the information about a file [141]. Last accessed time of a file or i_atime and last modified time of a file or i_mtime are attributes in an inode. A pointer member of *file* structure called f_inode points to an inode [135].

In [142], i_atime gets updated when one opens a file but also when a file is used for other operations like grep, sort, cat, head, tail, and so on. On the other hand, i_mtime gets updated when one modifies a file. Whenever she updates the content of a file or saves a file the i_mtime gets updated. Thus, for the content of the history of a critical file in a domU in Table 5.1, when a process reads a file (see the last column of rec_no 3), i_atime is t3 (see column 5 of rec_no 3), and is a timestamp of this event. When a process writes to a file (the last column of rec_no 2), i_mtime is t2 (in column 5 of rec_no 2), and is a timestamp of this event. Both i_atime and i_mtime are critical attributes for the history of a critical file. This section discusses how to obtain both of them.

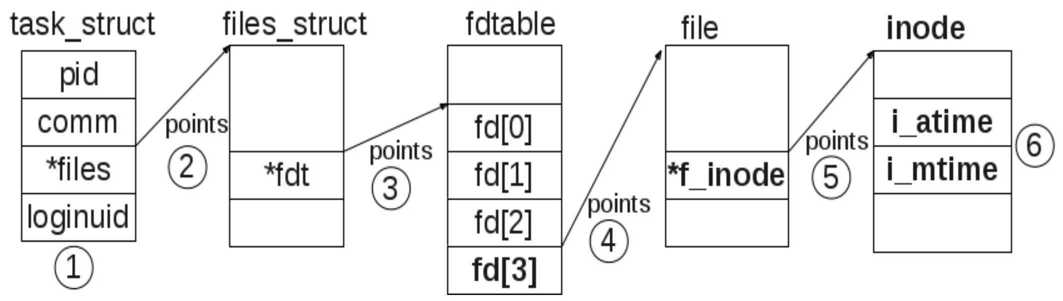


Figure 5.9: How to obtain information from inode

Figure 5.9 shows how to get i_atime and i_mtime, discussed in the lists below.

- The routines of Step 1 to 4 in the figure is the same as those in Figure 5.6.

- In Step 5, the logger follows a pointer f_inode to get the inode structure.
- In Step 6, the logger can obtain i_atime and i_mtime, which are members of the inode structure.

5.7.6 How to obtain the creation and deletion records of a critical file

This subsection explains how to obtain creation and deletion record of a critical file.

There are two types of critical files that a customer owns in his diskU:

1. uploaded critical files or files that a customer transfers from his local machine to his domU/diskU
2. newly created critical files or files that he newly creates in his domU/diskU, for instance by using appU.

Section 5.7.2.1 discusses the introspection system architecture, see Figure 5.3. This architecture comes with a condition. It is when the logger (a white box in dom0 user level in the figure) needs to capture history information of a critical file, the logger already knows the critical file's name such as "s.txt". The name will be in the memory space (read_mem in Figure 5.3) when any process accesses this file. This name will also be used as a trigger by the logger to capture which process is accessing this file, as discussed in Section 5.7.2.1.

Discussion of the creation and destruction record of both uploaded critical files and newly created critical files is presented in the lists below.

1. **A creation record of a critical file.**

   - To obtain a creation record of an uploaded critical file:

     Considering the first record of a critical file in our example of the history of a critical file in a domU, see Table 5.1. The content in the first record is the evidence of when a critical file was first created in diskU.

From the condition, a logger knows the name of a critical file. This name will be used as a trigger when the logger captures the history of a critical file. When a critical file is an uploaded file from local machines to a domU/diskU, the creation time (as t1 in rec_no 1 column 5 of Table 5.1) of this file can be either an i_atime or i_mtime.

Firstly, it is an i_atime (as discussed in Section 5.7.5), when a process reads this file for the first time in domU. Secondly, it is an i_mtime when a process writes this file. Thus, for rec_no 1 of Table 5.1, this is a record to indicate the creation of a file, and all the necessary information (including t1) of this record can be obtained by the logger. This is because we can consider the first record of an uploaded file as when the first time this file is accessed after the file is uploaded into domU.

However, the operation name of the operation on this file (the last column of rec_no 1 in Table 5.1) can be either read or write, instead of create. When the operation name is read, t1 (column 5 of rec_no 1 in Table 5.1) is i_atime. When the operation name is write, ti 1 is i_mtime. This is because, as discussed in Section 5.7.5, when a process reads a file then i_atime is a timestamp of this event, and when a process writes to a file then i_mtime is the timestamp of this event.

- To obtain a creation record of a newly created critical file:

  Following the same condition of our logging, which is the logger knows the file name, thus a customer has to create a new critical file in domU and allows the logger to know the file name of this new file. Therefore, the creation record of this new file is when the first time this file is accessed after it was created by this customer. Then the logger can capture the history information as discussed above. The creation record of a newly created critical file can be considered the same as the uploaded file in all aspects. Thus, the logger can then capture the history information for this file the same as discussed above.

2. **Destruction record of a critical file.** This discussion can be applied to both uploaded critical files and newly created files. For the destruction

record (rec_no N, in Table 5.1), we assume that a critical file will be deleted by an application or appU. For example, to delete a file name 's.txt' in the Linux system, the command to delete this file is *'rm s.txt'*. For this command (*'rm s.txt'*), s.txt is a parameter or argument of the rm command. The rm command is used for removing files or directories [143].

Thus, the logger needs to use a name 'rm' as a trigger to capture a parameter s.txt after the rm command. This capture can be achieved the same as for a mail command argument, as discussed in Section 3.4.2. To obtain the destruction record (rec_no N, in Table 5.1), the logger should capture the process id or p_id, process name or p_nm of rm command, the id of a user who runs this rm command or p_ownerId, and the time when this command is captured, see tN in column 5 in the table.

The operation name (the last column in the table) is fixed as 'delete' because we know that the rm command is for used deletion. It should be possible to capture file deletion that are performed by other commands or applications that have the ability to delete a file the same as the method for rm command.

5.7.7 To obtain the process behaviours logs through the introspection approach

This subsection discusses how to obtain process behaviours logs through the introspection approach. This includes obtainment of process behaviour logs when a process carries out an operation on a file, and the method to obtain process creation and destruction records.

To obtain process behaviour log when a process carries out an operation on a file. Table 5.3 shows the contents of process behaviour logs of a process or appU. Each record from the records rec_no 2 to rec_no N-1 is evidence of when a process has executed an operation on a file. When comparing the content of the history of a critical file in a domU (Table 5.1) with the content of process behaviour logs of a process or appU (Table 5.3), it can be seen that almost all the contents of both tables are the same.

The history of a critical file is file oriented, focusing on which process accesses this particular file such as s.txt, thus the file name is fixed, see Table 5.1 which does not show the file name because the file name is fixed as 's.txt' for example. The process log is process oriented, focusing on what this particular process is doing for example which file it has access to, rec_no 2 to rec_no N-1 in Table 5.3. Thus the process name is fixed as appU, see Table 5.3 column 3.

In Section 5.7.1, to obtain the history of a critical file, the logger (in the introspection approach or Figure 5.3) pinpoints a particular critical file name such as s.txt, thus the file name of the critical file is a main trigger for the logger to capture the history information. Similarly, this logger can be modified to pinpoint a process name to obtain process logs. The process name will be a main trigger for the logger to capture the process log information. Thus, the method to obtain process logs is as the same as the method to obtain history of critical files but different triggers.

To obtain process creation and destruction records. In Table 5.3, the content in the first record or rec_no 1 is the evidence or log that a process (with its id as 4001 or rec_no 1 column 2, and its name as appU or rec_no 1 column 3) is created by a user whose id is 1001 which is rec_no 1 column 4. The last record or rec_no N is the evidence of when this process or appU was terminated by a user id 1001 which is rec_no N column 4.

Section 5.6.3 defines the process of creation as when an appU is executed, which causes a live process name 'appU' to exist in memU. The section also defines the process of destruction as when an appU is terminated, normally when all of its instructions have been executed. This causes a live process name 'appU' to disappear from memU.

Section 5.7.2.3 discusses Linux kernel data structures for virtual memory organisation of processes and a list of processes, see Figure 5.5. The logger (a white box in dom0 user level in Figure 5.3) can be modified to capture both process creation and destruction records.

From Table 5.3, we assume that a logger is capturing the process creation or rec_no 1 and process destruction or rec_no N. To capture the creation of a process, the logger needs to access the list of all processes, see Figure 5.5. It then keeps checking the list until a target process such as read appears in the list. When read appU appears in the list, the logger can capture the detail of this process. The detail includes the process id and name of the read process, the id of a user who runs the read process, and the time (as t1, in rec_no 1 column 5 in Table 5.3) that the logger first found this process in the list. Thus, we assume that t1 is the creation time of this process.

For the destruction period, all information of this record is as same as rec_no 1 in Table 5.3 except the time tN which is rec_no N column 5 in Table 5.3. The time tN can be obtained by the logger checking the list until the same process name as the one in rec_no 1 (in this case the name is 'read') disappears from the list. Thus, we assume that tN is the destruction time of this process.

5.8 Interception

Another approach to obtain the history of critical files is **domU system calls interception approach**, for short-interception. This section mainly discusses how this approach can be used to obtain the history information.

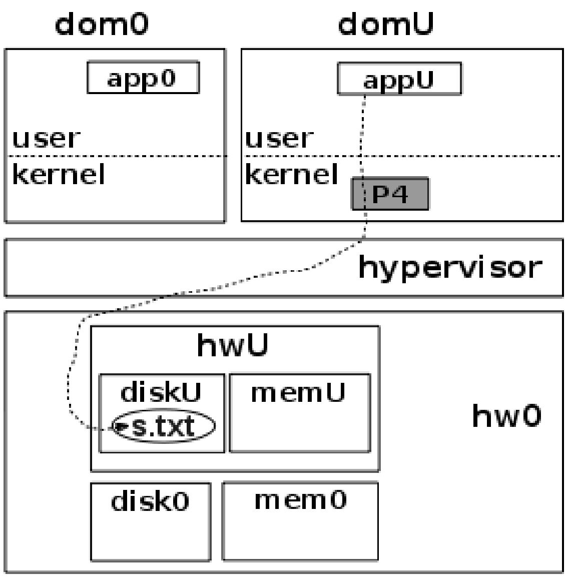


Figure 5.10: Interception logging solutions

As discussed in Section 2.12, only the kernel can access hardware such as disks, thus a process has to use system calls when it wants to access the hardware [144]. The system calls provide the interface between the user and kernel spaces [2]. For example, if a process wants to read the contents in a file, called s.txt, it has to issue at least three system calls which are to open the file, to read the file, and to close the file [144]. For example, from Figure 5.10, the dot-arrow-line represents an example of when appU reads s.txt

Considering the capture of history information of a critical file in domU, the interception approach is to intercept the file operations then to record the intercepted information as log files. From Figure 5.10, when appU reads s.txt (the dot-line represents an example), this appU needs to issue a 'read' system call via domU kernel. Thus, the main method to intercept domU's file activities (e.g., when appU is reading s.txt) is placing interceptors in the domU kernel level as P4 in Figure 5.10. These interceptors can intercept system calls that are invoked by appU.

[43] argues that some system calls are execve, fork, exit, read, readv, write, writev, mmap, open, pipe, and the kernel operation drop_inode, and these system calls are sufficient to capture the rich ancestry of relationships between Linux files, pipes, and processes, for instance the relationship information when a process reads a file. [30] also argues that placing interceptors in the domU kernel level can intercept information of relationship between a process and file, for example, where a process executes actions (include create, read, write, and delete) on a file.

Thus, domU system calls interception is an approach that requires the logger (called interceptor) to intercept domU system calls to tracks the creation, access, and destruction of processes, files, pipes, and sockets [43]. Thus, this approach can collect information of a process that opens a file or history of a critical file. In this thesis, the terms domU system calls interception and interception are used interchangeably.

How interceptors can obtain process behaviour logs. To obtain the history of a critical file, the interceptors such as P4 in Figure 5.10 can pinpoints a particular critical file name such as s.txt, thus the file name of the critical file is a main trigger for the logger to capture the history information. Similarly, these interceptors can pinpoint a process name to obtain process behaviour logs. Thus, the process name will be a main trigger for the interceptors to capture the process log information.

5.9 Discussion of introspection and interception approaches

This section discusses some aspects of both the introspection (Section 5.7.1) and interception (Section 5.8) approaches. The lists below are the discussions including: security aspects of introspection and interception which need to be analysed, helper components in introspection and interception approaches, the combination of introspection and interception, ideal logging solutions regarding the smallest TCB to produce the history of a critical file, and the failure of implementation for the ideal solution.

- **Security Analysis.**

  Security aspects of the introspection architecture (which deploys P1 in dom0 user level, Figure 5.3) and the interception architecture (which deploys P4 in domU kernel, Figure 5.10) can be analysed. Both architectures should have different security aspects. As a result, they may need different solutions to deal with their individual security issues. For example, the integrity of P1 and P4 above should be different. This is because they are deployed in different locations in the IaaS architecture.

  For example, locating P1 in a dom0 user level is a security risk because the providers or insiders may maliciously modify P1's code to produce contents of log files or F3, which benefit themselves, and the complete security analysis is in Section 5.10. P4 has different security concerns compared to P1 above. When P4 is in the domU user level, it could be at risk as well. This is because the owner of a domU can tamper with P4's code, as argued by [30; 39; 78; 96; 97].

These comparisons are very critical and need to be considered before deployment of introspection and interception in the production system. However, this is out of the scope of this research.

- **Helper components.**

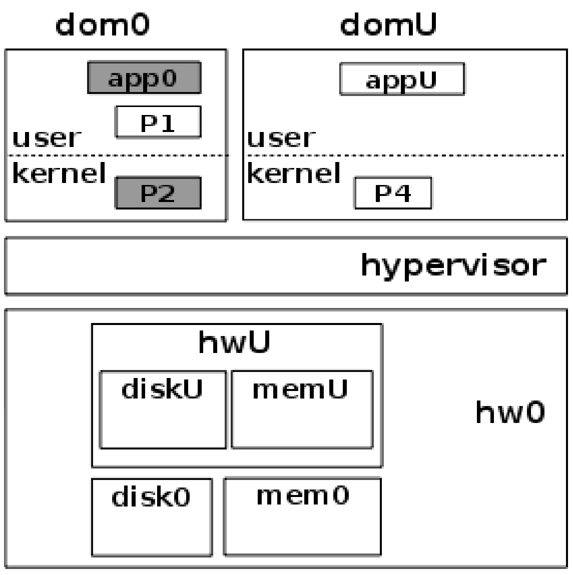


Figure 5.11: Helper components (app0 and P2)

Helper components can be app0 and P2 or the shaded boxes in Figure 5.11. Both introspection and interception may need these components to achieve the logging tasks of main components such as P1 (introspection) and P4 (interception). For example, app0 are deployed in [30; 39; 43; 44], and P2 are also deployed in works [30; 43; 44].

Helper components can perform a variety of logging-related tasks. App0 is in the dom0 user level, and could cooperate with dom0 user level logging processes such as P1 in Figure 5.11 to achieve logging tasks. For example in [39] which uses introspection, app0 is used for calling P1 (in case P1 is a library of tools such as libVMI) in order to access memU.

P2 is in the dom0 kernel level, and can cooperate with other logging processes such as P4 in Figure 5.11, which is in the domU kernel. For example, PASSXen [43] which is an interception has an analyser or P2 as a helper component in dom0 kernel level to communicate with P4 to process log records. However, helper components can be in the domU kernel level as well, such as in HP Flogger [30]. These helper components need to be considered because they may affect the security analysis and performance of logging systems.

- **The combination of introspection and interception.**

  If the combination of introspection and interception is needed to mitigate the risks associated with the CSA threats, the performance and security of this new approach may be more complicated. For example, the combination will cause more distributions of Px to many places in the IaaS architecture, which will affect the security and performance logging systems. However, it is possible to combine both approaches to mitigate the risks associated with CSA threats, regarding security and performance of logging systems which are deployed in the IaaS.

- **Smallest trusted computing base or TCB.**

  Introspection and interception approaches involve TCB concerns. [38] argues that the TCB size of a software system (a logging system is also a software system) can be used to evaluate the trustworthiness of that system. [39] state that for any software system, the size of the system's TCB should be as small as possible. Section 3.5.2.8 has already discussed the smallest TCB size of a logging system.

  Deploying all logging related components in a hypervisor. For interception, the smallest TCB can be achieved when moving all logging processes or components into the hypervisor. To compare the TCB of introspection with interception, the interception (P4 in domU kernel level in Figure 5.10) will have less TCB than the introspection (P1 in dom0 user level in Figure 5.3) when all the interceptors and related logging processes (P4) are moved and implemented in a hypervisor. Thus, it should be an ideal consideration that we can deploy logging processes as P5 in the hypervisor as shown in Figure 3.7. P5 should collect a history of the critical file in domU.

- **The failure of implementation to place all logging processes as P5 in Xen with the interception approach**

  We decided to implement the ideal solution to achieve the smallest TCB with the interception approach, see Figure 3.7. We need to place all logging processes into Xen. We followed the work of [93] to implement this approach. Basically, they modified traps.c file (located in /xen/arch/x86/) in Xen. They demonstrate how to obtain: system call number such as 78; system call name such as gettimeofday; and domU id such as 8. However, they do not discuss how to capture a critical file name that is associated with a process which is accessing the file. We investigate some system calls that relate to a file, such as sys_read, sys_write, sys_open, and sys_close [145]. However, we managed to get only domU id, and some system call ids which did not relate to a file.

  This approach needs to place all logging processes as P5 in Xen with interception, and we realise that this approach is complicated. PASSXen [43] also do not publicly publish the code of their work, which is necessary to deploy this approach. With time limitations, we started to investigate the introspection approach. The next chapter will demonstrate how to attain the history of critical files using the introspection approach (Figure 5.3). Although, the introspection approach does not yield the smallest TCB size, we succeeded in implementing the proposed logging system based on this approach. The proposed system yields less TCB compared to some previous work, as will be discussed in the next chapter.

5.10 The Security Analysis of the Proposed Logging System

This section discusses the security analysis of the system architecture of the introspection approach (Figure 5.3). This architecture is used as the architecture of the proposed logging system in the next chapter.

The proposed logging system's architecture in the implementation is systematically designed based on the generic logging components. Therefore, it inherits the advantages of the generic logging components. For example, one of the advantages is an architecture based on the generic logging components can be used as a tool to analyse the security of logging processes and log files, which are distributed across the IaaS visualisation environment. Thus, one can analyse the security of the logging system architecture in the implementation.

This thesis did not aim to provide secure logging systems yet in this stage. This means that the proposed system (Figure 5.3 as will be discussed in Section 6.1) can be tampered with, as discussed in Section 3.4.2.4. The system architectures of the logging system discussed in Section 3.4.2.4 and one in Figure 5.3 are the same. We provide an approach to systematically analyse the security of logging systems before deploying them in the IaaS real world productions. Section 3.4.2.4 has already discussed how to analyse the security of our logging system in the spamming case study.

DomUs cannot tamper with the components of the proposed logging system. Our implementation deploys the logger and libVMI in dom0, and the log files or F3 in hw0. The advantage of doing this is that domUs cannot tamper with these components. In contrast, when deploying the components inside domU, this allow the owner of this domU to tamper with these components, as argued by [30; 39; 96]. However, deploying the logging processes in dom0 also should consider the other security aspects as discussed below. The solutions for these security issues are out of the scope of this thesis.

Firstly, **the security analysis of the log files**: we use the logging system architecture (Figure 5.3) as a tool for this analysis. For example, the relevant security question is how to ensure the integrity of the log files that are stored in disk0 which is fully owned and controlled by a provider. The provider may maliciously learn about, or alter the log files. Secondly, **the security analysis of logging processes**: the next relevant security question is how can an auditor ensure the integrity of the logger or libVMI, which is run by the provider in dom0. Locating these components in this location can be a security risk. This is because the provider may maliciously modify the logger's code to produce contents of log files, which benefit himself. We discuss the questions above as an example of the analysis of the proposed logging system.

5.11 Conclusions

This chapter addressed research Gap 1 (lack of systematic approaches to build logging systems in IaaS), in particular, research Gap 1 (a) which is the lack of simultaneous consideration of both customers and providers. Importantly, it addressed research Gap 2 which is that research relating to logging in IaaS only focuses on system-centric logs.

This chapter provided two main contributions. The first contribution is the proposed solution to collect file-centric logs rather than system-centric logs. This contribution addresses research Gap 2. There are some logging solutions that emphasise file-centric logs with an interception approach. However, the prototype implementation of the proposed logging system (in the next chapter) can be an alternative approach to collecting file-centric logs to enhance accountability in IaaS. This approach advocates the introspection of customer VM's memory from dom0. The introspection traverses the kernel data structures in the memory.

The next contribution is the analysis of how CSA threat 1 affects both customers and providers simultaneously, and the proposed logging solutions that assist in mitigating the risks associated with the threat for both customers and providers. This contribution addresses research Gap 1 (a). The analysis illustrates how CSA threat 1 such as mis-usage of customer VMs can affect both sides.

The next chapter is prototype implementation of the proposed logging solutions that can collect the file-centric log history of customers' critical files. An architecture of the prototype is simple and cheap ways for the design and development (the main benefit of the simple system pattern in Figure 4.7 Section 4.2.2.1) as discussed in Section 4.2.3 from Chapter 4. The architecture also facilitates enforcement of a security policy of a system itself, which is the main benefit of the security facilitator pattern (Figure 4.8 Section 4.2.2.2).

Chapter 6

Implementation

This chapter is prototype implementation of the proposed logging solutions that can collect the file-centric log history of customers' critical files which are discussed in the previous chapter. An architecture of the prototype is simple and cheap ways for the design and development (the main benefit of the simple system pattern in Figure 4.7 Section 4.2.2.1) as discussed in Section 4.2.3 from Chapter 4. The architecture also facilitates enforcement of a security policy of a system itself, which is the main benefit of the security facilitator logging pattern (Figure 4.8 Section 4.2.2.2 from Chapter 4).

This chapter addresses three research gaps. The first one or research Gap 1 (b) is the lack of concern for reducing the trusted computing base or TCB size of a logging system. The second one or research Gap 2 is an issue that research relating to logging in IaaS only focuses on system-centric logs which is the hardware layer log, such as memory use, disk storage, temperature, and voltage. The third research gap or research Gap 3 is a lack of analysis of what the contents of the log file should actually be, and of how the contents can be used to deal with the real world CSA threats to benefit both sides in detail.

The chapter has three main contributions. Firstly, the design of our proposed logging system yields TCB size compared to previous work. This contribution addresses Gap 1 (b). All logging related components (such as an introspection tool that we have re-used, and a logger application) that have been deployed in building the proposed logging system in this chapter are inside dom0.

The TCB size of a logging system can be compared to a TCB size of another logging system, as discussed in Section 3.5.2.8. For example, the TCB size of a logging system, which includes all components (hw, a hypervisor, dom0, and domU) is bigger than the TCB size of another logging system, which includes only hw, a hypervisor, and dom0. The TCB size of our architecture includes only a hypervisor and dom0, not domU. In contrast, previous works place some of their logging-related components in domU; thus, the TCBs of their systems have to include domU apart from a hypervisor and dom0. Other previous work that have deployed the same introspection tool as in this chapter yields the same TCB as ours. However, they are not designed to log the history of critical files.

Secondly, the proposed system collects file-centric logs rather than system-centric logs. This contribution addresses research Gap 2. The prototype implementation of the proposed logging system can be an alternative to collect file-centric logs in order to enhance accountability in IaaS by domU's VM memory introspection approach (by traversing the domU kernel file structures) from dom0.

Moreover, the file-centric history logs can be associated with a process and files in a domU, for example, a record of a process P which reads file F. The proposed log files differ from previous work which focus only on system-centric logs such as the connection topology, bus speeds, and processor loads.

Lastly, this chapter presents how the results from the proposed logging system assist in mitigating the risks associated with threat 1 with nine incident scenarios. This contribution addresses research Gap 3. These scenarios illustrate how the results from the implementation can be used to form the history of critical files. These files can be used to assist in mitigating the risks associated with CSA threat 1 for a customer, when the customer is the only user in the domU or when she shares the domU with many users. We then discuss how the results can form process behaviour log files to assist in mitigating risks associated with threat 1 to benefit providers. This discussion includes how the process logs can be used to investigate the causes when the dom0 is compromised, and when attackers use domUs for spamming activities.

Thus, with systematic approaches, this shows that our proposed solutions can assist in mitigating the risks associated with real world IaaS issues and many scenarios. The proposed solutions also benefit both customers and providers.

The remainder of the chapter is structured as follows. Section 6.1 provides the design and implementation of a logging system to produce the history of critical files with an introspection approach. This includes a description of context of a domU which we want to capture the history information from, a discussion of the design and implementation of the prototype of the proposed logging system, and an explanation of running of the experiment in the implementation.

Section 6.2 discusses how the results can be used to form the history of critical files to assist in mitigating the risks associated with threat 1 to benefit customers. This includes how the formed history logs assist in analysing nine incidents in Alice's domU.

Then, Section 6.3 discusses how the result can benefit providers. This includes a discussion of how the results from the implementation can assist in mitigating risks associated with threat 1 for the providers such as dealing with the compromising of dom0, and with spamming activities.

In Section 6.4, the proposed system from the implementation is compared to related work. This includes the comparison between the TCB size of the proposed system and TCB sizes of the logging and monitoring systems from previous works. Then Section 6.5 provides the performance measurement of our proposed logging system in terms of its accuracy in capturing logging information (e.g., a file name of a file that is read by a domU's application) from domU. Finally, the chapter is briefly summarised and concluded in Section 6.6.

6.1 Implementation

This section provides the design and implementation of a logging system to produce the history of critical files with an introspection approach. It begins with a description of context of a domU we want to capture the history information from this domU. Then, it discusses the design and implementation of the prototype of the proposed logging system, and explains running of the experiment in the implementation.

As discussed in Section 3.5.2.8 that if all logging processes of a logging system are in only a hypervisor, the TCB size of this system includes only hw0 and hypervisor. Then, this TCB size is the smallest one. As discussed by the last list item in Section 5.9 that to achieve the smallest TCB size (by placing all logging processes as P5 in hypervisor) is complicated in the interception approach. We started to investigate the introspection approach.

This section demonstrates how to get the history of critical file using an introspection approach. Although, the introspection approach does not yield the smallest TCB size, we have succeeded in implementing the proposed logging system based on this approach. The proposed system yields less TCB compared to some previous works, as will be discussed in Section 6.4.

6.1.1 Context of A DomU in the Implementation

This subsection discusses the environment setting in a domU for the implementation. We already discussed the environment of a domU where the introspection approach can operate in Figure 5.2 in Section 5.7.1. We use this setting environment for this implementation. In the figure, we assume that Alice rents a Linux domU. In Figure 5.2, she has a critical file or s.txt which is in the ellipse in diskU. The actual location of s.txt is in the '/home/alice/' directory.

Section 5.6.3.1 already discussed how a C program becomes an appU or a process. In this experiment, read application is an appU, see the rectangle in domU user level in Figure 5.2. This read appU is read.c which is a C program. The program is stored in domU/diskU. When Alice compiles read.c which is in '/home/alice/' directory in diskU, the 'read' executable program of application is created and stored in this directory. Alice can run this program by command './read' from '/home/alice/' directory. The name 'read' is the appU's name and also the process name of the appU. Alice can run this application to read s.txt, see the dot-arrow line in Figure 5.2. Read application routines are to:

1. open s.txt file,
2. read the file,
3. close the file, and
4. be terminated.

Read_mem or the ellipse in memU in Figure 5.2 is the memory space of this read appU or process when it is reading s.txt. This memory space holds all information we need to record, as discussed in Section 5.7.1. The first piece of the information is a file name of s.txt. The rest is the history information of s.txt shown in rec_no 1 in Table 5.2, listed below:

1. a process Id as 4624 of read appU,
2. a process name of read appU as read, and
3. an owner Id of read process as 1002 which is Alice's user Id in this domU.

The logging system in this implementation records the history of critical files (rec_no 1 in Table 5.2, as discussed above), and then stores them in a log file.

6.1.2 System Architecture of the Proposed Logging System

This subsection describes the system architecture of the proposed logging system. The logging system in this implementation records the history of critical files (rec_no 1 in Table 5.2, as discussed in Subsection 6.1.1), and then stores them in a log file.

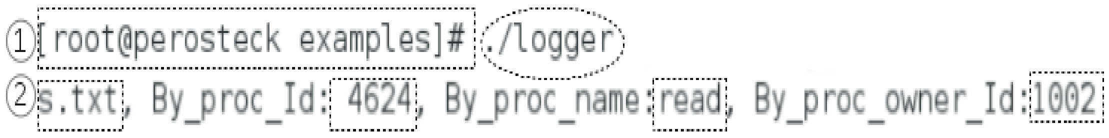
Section 5.7.2.1 has already discussed introspection system architecture or Figure 5.3. We use this introspection system architecture for this implantation in this chapter. This is because the architecture is simple and cheap ways for the design and development (the main benefit of the simple system pattern in Figure 4.7 Section 4.2.2.1) as discussed in Section 4.2.3 Chapter 4. The architecture also facilitates enforcement of a security policy of a system itself, which is the main benefit of the security facilitator logging pattern (Figure 4.8 Section 4.2.2.2 from Chapter 4).

Thus, Figure 5.3 is the system architecture of the proposed logging system. The main components of the architecture are:

1. an app0 or logger (the white box in dom0 user level in Figure 5.3) which is an executable program (written in C programming language) located in '/root/examples' directory in dom0/disk0,

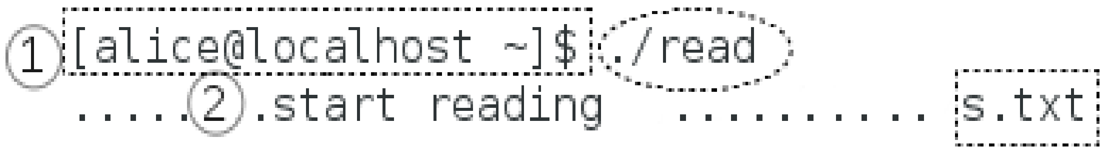
2. P1 or libVMI [79] (the shaded box in dom0 user level in Figure 5.3) which is a C library that is installed in this dom0 system, and
3. F3 as a log file which is a text file located in '/root/examples' directory in disk0.

6.1.3 The routines of the logger



```
[root@perosteck examples]# ./logger
s.txt, By_proc_Id: 4624, By_proc_name:read, By_proc_owner_Id:1002
```

Figure 6.1: The logger running in dom0 to record the history of the critical file s.txt



```
[alice@localhost ~]$ ./read
.....start reading ......... s.txt
```

Figure 6.2: The read command running in domU and reading critical file s.txt

This subsection discusses the routines of the logger in the proposed system architecture.

The basic idea of the logger's main task is to record the user id or p_ownId of a user who logs in to a domU, when this user uses any appU to access a critical file. Figure 6.1 and Figure 6.2 assist in explaining the routines. Both figures present how to run the experiment for this implementation and the output or result of the running. A circle with a number in both figures indicates a step of running a program. The circle also indicates a line number of output or result of the running of a program. A rectangle in the first line of each figure represents a Linux command prompt for a user to type in commands to run application. An ellipse in the first line of each figure is a Linux command that a user types in to run application (e.g., logger or read application). A rectangle in the second line of each figure represents an important output of a program.

Assumptions of the logger. It is assumed that:

1. the logger knows the name of the process (which is the 'read' process) that will read s.txt,

2. the logger knows that it will capture a file name called 's.txt'.

Thus, the process name 'read' as trigger 1 and the file name 's.txt' as trigger 2 will be used as the triggers of the logger to capture the history of critical file s.txt, as discussed below. Note that, when the read command starts reading s.txt or the 2nd line in Figure 6.2, this is when a memory space of read process (see Figure 5.2, *read_mem* or the ellipse in memU) contains Linux kernel data structures for virtual memory organisation of the read process or Figure 5.6. All information that the logger needs to capture is in this memory space.

The lists below are the routines of the logger.

1. The logger command as app0 runs in dom0, see the dot-ellipse in the 1st line of Figure 6.1. After this command starting, it keeps checking memU until the 'read' command as trigger 1 is performed and exists in memU.

2. When a read command is performed in domU (see the dot-ellipse in the 1st line of Figure 6.2), the logger pinpoints this command then keeps waiting until the 'read' command or process reads s.txt which is trigger 2.

3. When the read command starts reading s.txt (the 2nd line in Figure 6.2), the logger immediately extracts the necessary information, see 2nd line of Figure 6.1. It extracts the file name or a string "s.txt", see the first dot-box in the 2nd line of Figure 6.1. Then the logger extracts the history information as in rec_no 1 in Table 5.2, listed below:

   - process id as 4624, the second dot-box in the 2nd line of Figure 6.1,
   - read process name as 'read', the third dot-box in the 2nd line of Figure 6.1, and

- the id of the owner of read command as 1002, the last dot-box in the 2nd line of Figure 6.1.

  Note that, the methods of extraction of the information (file name as s.txt, process Id as 4624, process name as read, and process owner Id as 1002) discussed above follow Step 1 to Step 8 as explained in Section 5.7.2.3.

4. After that, the logger writes this extracted information above to a log file as F3 in Figure 5.3.

We assume that a provider controls and can run the logger in dom0, and a customer or Alice can control and run the 'read' application in domU. However, we will run the logger and read application on behalf of both the customer and provider in this implementation.

6.1.4 The Experimental Environment

This subsection describes the experimental environment of a machine for the implementation. The experimental environment comprises a single physical machine as hw0. Its hardware configuration includes an Intel Core i7-860 64-bit CPU and 3980556 kB of main memory. The hw0 consisted of hypervisor that is Xen 4.3-unstable with a Fedora 17 dom0 running a 64-bit Linux kernel, version 3.7.9-104. Alice's domU is running a 64-bit Linux kernel version 3.8.5-201 for Fedora 18. It consisted of 738.6 MiB of main memory.

6.1.5 Running the experiment

This subsection explains the running of the experiment in the implementation. To run one set of the experiment, this entails running both the logger and read appU. The steps below is to run one set of the experiment in this implementation.

- Step 1 (the circle with number 1 in Figure 6.1), with dom0 root privilege, we run the logger, see the dot-eclipse in the first line of Figure 6.1. The command to run the logger is './logger', see the dot-ellipse in the 1st line of Figure 6.1. We run the command in '/root/examples' directory.

- Step 2 (the circle with number 1 in Figure 6.2), we run read appU, see the eclipse in the first line of Figure 6.2. The command to run read appU is './read', see the dot-ellipse in the 1st line of Figure 6.2. We run the command in '/home/alice' directory.

6.2 Discussion of the results for the customers

This section discusses how the results can be used to form the history of critical files to assist in mitigating the risks associated with threat 1 to benefit customers. This includes how the formed history logs assist in analysing nine incidents in Alice's domU.

Assumptions for the discussion. As discussed in Section 5.1.2, when a customer such as Alice or Carol rents a domU then she will get two accounts at the same time. They are a root user account and a standard user account. Then, Section 5.4 discussed the many reasons that her domU may be compromised according to threat 1. The lists below are the assumptions for the discussion in this section.

- After one's domU is compromised, we assume that an attacker in some manner is able to compromise her root user account and standard user account.
- Then, we assume that the attacker can gain access to her domU and can use appUs in this domU to conduct criminal activities. For example, the attacker may use appUs to maliciously access her critical files.

For the incident when the attacker maliciously accesses her critical files, the history of her critical files can be used as evidence to assist in discovering the incident. History of critical files is a file-centric log which traces files from the time they are created to the time they are deleted as discussed in [27; 45]. [43], [30], and [76] argue that file-centric logs are very important aspects to deal with the cloud problems.

We do not demonstrate the assumptions above. However, these assumptions are used in proposing file-centric logging solutions in the cloud. [30] and [76] consider these assumptions when proposing their logging solutions in the cloud. For example, [30] assumes that when a user 'Alice' creates a sensitive file and modifies this file, later, a user 'Bob' can read the file without Alice's permission. Bob can read the file, thus one of the consequences of Bob's ability to read the file can be that Bob can gain access to Alice's domU.

It is critical when Alice's user accounts are obtained by attackers. This is because the attackers may access Alice's critical file such as s.txt in diskU in Figure 5.2. Section 5.6.2.1 discusses how the history of critical files can assist in mitigating the risks associated with this threat to benefit customers. This section discusses in detail how the history of s.txt can assist in analysing malicious incidents in Alice's domU to mitigate the risks for the benefit of the customers. This thesis uses the term 'auditor' to refer to a person who can obtain and use the history of critical files to analyse incidents of misuse of domUs.

The results from the implementation (e.g., 2nd line of Figure 6.1) can be used to form the history of critical files (discussed in CASE 1 to CASE 9 below) which can be used to assist in mitigating risks associated with CSA threat 1 for the customers in many cases. CASE 1 to CASE 9 are based on how Alice uses her domU, including when she uses it as the only user in the domU or when she shares the domU with many users.

Section 6.1.2 discusses system architecture of the proposed logging system, see Figure 5.3. The logger (the white box in dom0 user level in Figure 5.3) can be modified to capture the history of critical file to be used for discussion in CASE 1 to CASE 9. The logger records a user id or p_ownId of a user who logs in to a domU, when this user uses any appU to access a critical file, as discussed in Section 6.1.3.

6.2.1 Analysing malicious incidents when Alice is a single user in a domU

This subsection provides analysis of malicious incidents when Alice is a single user in a domU.

rec_no	p_id	p_nm	p_ownId	time
101	4624	read	1002	t1
102	4002	read	1002	*t2
103	4002	read	1002	t3

Table 6.1: Parts of the fictitious contents of the history of s.txt

In the experiment, the domU is for a single user. However, Alice owns two accounts: the root which can run all appUs in the domU; and alice or an administrator. We assume that the contents of Table 6.1 is part of a complete history of a critical file, as shown in Table 5.1 in Section 5.6.1. However, Table 6.1 is a fictitious table to represent the history of a critical file.

Thus, Table 6.1 presents only rec_no 101 to 103 of the contents of the history of s.txt for the purposes of this discussion. The contents in Table 6.1 can be constructed from the results from 2nd line of Figure 6.1. Note that, in this table, time or column 5 presents the last accessed times of the file. An asterisk or * in the following tables indicates possible malicious events in domUs.

We use 'Alice' as the name of a person, and 'alice' as the user name of Alice. This condition applies for every person name and every user name. When the history of s.txt is available to Alice, she may audit s.txt. CASE 1 and CASE 2 below discuss how the history of critical files can be useful for analysing malicious incidents when Alice is a single user in a domU

Rec_no 101 in Table 6.1 is an example of a normal event when Alice logs in to her domU. Then, she runs her read appU (rec_no 101 in column 3) to access s.txt (the dot-arrow line in Figure 5.2) with her or root permission. The history of s.txt or Table 6.1 shows Alice's Id as 1002 in rec_no 101 column 4. However, when root's or alice's password of Alice's domU is compromised as an example of threat 1, rec_no 102 in Table 6.1 can represent suspected incidents related to her file, in the discussions in CASE 1 and CASE 2 below.

CASE 1 In rec_no 102 in Table 6.1, this record can present undesired access to s.txt. The reason can be that Alice has never accessed s.txt at t2 time, see rec_no 102 column 5. Thus, she may suspect that attackers may have accessed s.txt.

We assume that an interval between times tx and ty (e.g., from column 5 in Table 6.1, an interval can be between t1 and t2 or t2 and t3) is big enough for Alice to recognize that t2 (the attacker's event) is suspicious compared to t1 (time of Alice's event) or t3 which is time of another Alice's event. For example, on the same date such as 25th October 2013, t1 can be at 08.00, t2 is at 13.00, and t3 is at 17.00. Thus, Alice can distinguish times of her (t1 and t3) and the attacker's event (t2). This assumption also applies to discussion of times in CASE 2 to CASE 9.

rec_no	p_id	p_nm	p_ownId	time
101	4624	read	1002	t1
102	4002	read	1002	t2
103	4003	*otherApp	1002	t3

Table 6.2: Parts of the fictitious contents of the history of s.txt, for CASE 2

CASE 2 In rec_no 103 in Table 6.2 , this record can also be undesired access to s.txt. This is because Alice has never used otherApp (rec_no 103 column 3) to access s.txt. Thus, she may realise that attackers may be doing this. After pinpointing otherApp, in order to obtain more evidence for auditing, she may undertake further investigations.

6.2.2 Analysing malicious incidents when Alice is in a multiple users domU

This subsection provides the analysis of malicious incidents when Alice is in a multiple users domU, in CASE 3 to CASE 9 below.

This thesis presents two types of multiple users domU, as discussed in Section 5.1.2. The first is multiple users using a domU with one root. The second type is multiple users using a domU with more than one root. CASE 3 to CASE 7 are a discussion of multiple users operating in a domU with one root. CASE 8 and CASE 9 are a discussion of multiple users using a domU with more than one root.

CASE 3 This case explains how the history of critical file is useful for the environment of multiple users using a domU with one root. The domU environment in this case is multiple users operating in a domU with one root user and when Alice is a root user, as discussed in Section 5.1.2. The domU in this implementation can be configured to allow multiple users environment. In this case, it is supposed that there are three user accounts in Alice's domU, as listed below.

- The first one is a root user account.
- The second account is alice user. Thus, Alice owns two accounts: the root account and alice account.
- The third account is bob user which is a standard user. This account is created by Alice for Bob to log in to and share Alice's domU.

As discussed in Section 5.1.2, a standard user can only access his or her files in specific locations which are managed and provided by the system. For example, a directory of alice user to store her files can be '/home/alice'. Bob cannot access '/home/alice/' directory, but a root user may. As discussed in Section 5.1.1, a root user can access all parts or files of the system [105]. Thus, the root user in Alice's domU can access any file in '/home/alice/'. The root user can also execute any executable files in this directory, but bob user cannot. The executable file can be read appU which belongs to Alice.

As discussed in Section 5.4, this domU can be compromised by threat 1. Thus, attackers may obtain both Bob's and root's password. Then, they may log in to Alice's domU using Bob's account. Although the attackers can log in to Alice's domU with Bob's account, they cannot access '/home/alice/' directory, as discussed above. However, if they know root's password, then they can run Alice's read appU to illegally access s.txt with root permission. This illegal incident is significant.

Alice may discover the incident discussed above by auditing Table 6.3 which presents parts of the fictitious contents of the history of s.txt in the multi-user domU. The evidence can be seen in rec_no 104 of the table. This record can be a suspect incident showing how Bob as id 1003 (column 4 of rec_no 104) accesses Alice's s.txt. Although the attacker uses the same appU as read (column

rec_no	p_id	p_nm	p_ownId	time
.	.	.	.	.
104	4004	read	*1003	t4
.	.	.	.	.

Table 6.3: Parts of the fictitious contents of the history of s.txt in a multi-user domU

3 of rec_no 104) which is normally used by Alice to access s.txt, the log still shows Bob's Id as 1003, which is not an owner of s.txt. Consequently, Alice could eventually discover this suspicion from the history of s.txt. Thus, the history information in Table 6.3 can be evidence to assist in analysing undesired incidents inside domUs. As a result, this can assist in mitigating the risks associated with threat 1 to benefit customers.

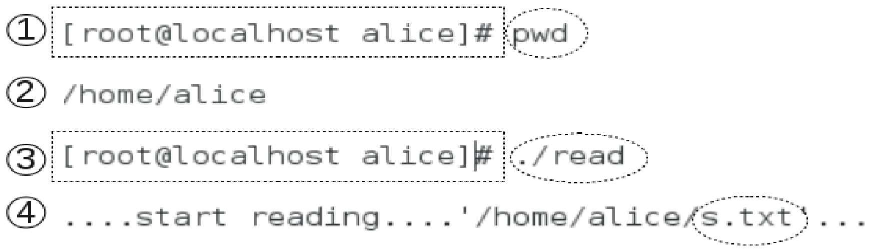
CASE 4 This case discusses how the history of critical files can be useful for multiple users using a domU with one root, and when Alice is a standard user not a root user. In this case, it is assumed that a domU is owned by Carol, who owns a root user account and the carol user account, and that Alice and Bob are standard users who shares Carol's domU. Thus, there are four user accounts in this environment of Carol's domU as listed below. Note that, this environment is for discussion of CASE 4 to CASE 7.

- The first is a root user account.
- The second account is the carol user account with id 1000.
- The third account is the alice user account with id 1001, which is a standard user. This account is created by Carol for Alice to log in to and share Carol's domU.
- The fourth account is the bob user account with id 1002, which is a standard user. This account is created by Carol for Bob to log in to and share Carol's domU.

Rec_no 201 in Table 6.4 is a normal activity of Alice with her s.txt, when she runs her read appU to access s.txt with her own privilege. The log file records Alice's id, 1001.

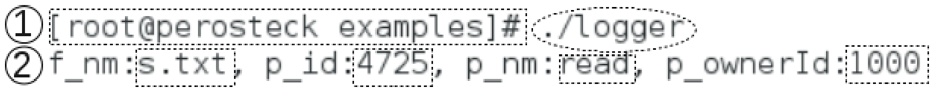
rec_no	p_id	p_nm	p_ownId	time
201	4624	read	1001	t1
202	4725	read	*1000	t2
203	4726	*rootRead	*1000	t3
204	4727	read	*1002	t4
205	46268	read	1001	*t5

Table 6.4: The content of the history of critical file f (s.txt)



```
① [root@localhost alice]# pwd
② /home/alice
③ [root@localhost alice]# ./read
④ ....start reading....'/home/alice/s.txt'...
```

Figure 6.3: Running read command to access s.txt in domU



```
①[root@perosteck examples]# ./logger
②f_nm:s.txt, p_id:4725, p_nm:read, p_ownerId:1000
```

Figure 6.4: Record of read command running to access s.txt with Calor privilege

Thus, CASE 4 can be rec_no 202 in Table 6.4, which is a malicious incident. This record can be a suspect incident that why and how Carol with id 1000 (column 4 of rec_no 202 in Table 6.4) accesses Alice's s.txt. This incident may be because Carol's domU is compromised. This compromise can be one of the forms of attacks enclosed in threat 1, as discussed in Chapter 5.

Assuming the attackers obtain: a root user account including root's user name and password[1]; and the carol user account including Carol's user name and her password, the following events below are assumed.

- The attackers may use carol account to log in to Carol's domU.
- The attackers open a terminal, request root accesses in the terminal with the stolen root account.
- In the terminal, they go to '/home/alice' directory. In Figure 6.3, Step 1, 'pwd' in the ellipse is a command to show a current directory (Step 2, currently is '/home/alice').
- The attackers run Alice's read appU (the ellipse in Step 3 in Figure 6.3) to read s.txt in '/home/alice/' directory (Step 4) with root privilege.

However, the logger (in the ellipse in Figure 6.4) records Carol's id or 1000 (the last dot-rectangle in line 2 of Figure 6.4) as evidence of the events when s.txt is accessed, see in Table 6.4 rec_no 202 column 4.

Other log information of these events are in the first, second, and third dot-rectangles in line 2 of Figure 6.4. They are: s.txt file name as a string "s.txt"; a process id as 4725; and a name of read appU as a string "read". The events are shown in Table 6.4 rec_no 202. Note that, we omit the time attribute or the last column of Table 6.4.

CASE 5 This case can be rec_no 203 in Table 6.4, which is a malicious incident. This record can be a suspect incident that why and how Carol as id 1000 (column 4 of rec_no 203 in Table 6.4) accesses Alice's s.txt using rootRead appU which is shown in column 3 rec_no 203 in Table 6.4. This incident may be because

[1] In Section 5.1.1, we defined a user account as composed of two pieces of information: i) a user name and ii) a user password.

Carol's domU has been compromised. This compromise can be one of the forms of attacks enclosed in threat 1, as discussed in Chapter 5.

Assuming the attackers obtain: a root user account including root's user name and password; and Carol's user account including Carol's user name and her password. The events to access s.txt by rootRead appU are as same as the steps in the lists discussed in CASE 4 above. However, in CASE 5 it is assumed that the attackers use rootRead appU to access s.txt rather that using read appU as in CASE 4.

However, the logger records Carol's id as 1000 (rec_no 203 column 4 Table 6.4) as evidence of the events when s.txt is accessed by rootRead, see in Table 6.4 rec_no 203 column 3. Other log information of these events is a process id of rootRead appU, which is 4726, see in Table 6.4 rec_no 203 column 2. We omit the time attribute or the last column of Table 6.4.

CASE 6 Rec_no 204 in Table 6.4 can represent undesired access to Alice's s.txt. This case may be because Carol's domU may subsequently be compromised. This compromise can be one of the forms of attacks enclosed in threat 1, as discussed in Chapter 5. Then, Bob's account and the root account is compromised by attackers. After that, the attackers log in to the domU using Bob's account, and run Alice's read appU with root privilege.

However, id of a user that is recorded by the logger is Bob's id as 1002 which is rec_no 204 column 4 Table 6.4. The id is used by the attackers to gain access to this domU, and they can access Alice's s.txt. Thus, Alice can discover this incident from Table 6.4 and may undertake further investigation.

CASE 7 Rec_no 205 in Table 6.4 can be undesired access to Alice's s.txt. This case may occur because Carol's domU is compromised (threat 1, as discussed in Chapter 5), then Alice's account is subsequently compromised. Then, attackers log in to domU using Alice's account and run Alice's read appU using her privilege. Note that, it is no need to use root privilege to run read appU to access s.txt because Alice is the owner of both of them.

Rec_no 205 in Table 6.4 can refer to undesired access to Alice's s.txt. The reason can be that Alice has never accessed s.txt at t5 time, which is rec_no 205 column 5 Table 6.4. Thus, she may suspect that attackers may have accessed s.txt.

CASE 8 This case discusses how the history of critical files is useful for discovering malicious incidents in a domU when Alice is in the environment of multiple users using a domU with more than one roots. Section 5.1.2 has already discussed how to set this environment in domU. Note that, this environment is for discussion of CASE 8 to CASE 9.

In this case, it is assumed that a domU is owned by Alice. Thus, she owns a root user account (called root) and the alice standard user account. Then, she adds a new root user account called 'rootusr' using approaches described in [108]. Therefore, this domU has two root user accounts (called root and rootusr), and rootusr has the same privileges as root including full access to all files in this domU. Both Alice and the new rootusr can log in to this domU with their own individual and separate root password. All user accounts are listed below.

- The first is a root user account called 'root' assumingly with id 0.
- The second account is the alice user account with id 1000.
- The third one is a root user account called 'rootusr' assumingly with id 1003.

Alice's domU may be compromised (threat 1). Then, rootusr's account and Alice's account may be further compromised by an attacker, and the attacker logs in to this domU using Alice's account. After that, the attacker runs Alice's read appU or dummyAppU to access s.txt with rootusr's privilege.

rec_no	p_id	p_nm	p_ownId	time
302	4725	read	1000	*t302
303	4726	*dummyAppU	1000	t303

Table 6.5: Parts of the fictitious contents of the history of s.txt, for CASE 8 and CASE 9

Rec_no 302 in Table 6.5 can represent undesired access to Alice's s.txt. The reason can be that Alice has never accessed s.txt at t302 time (rec_no 302 column

5 Table 6.5) though the id in the table is Alice's id. Thus, she may suspect that attackers may have accessed s.txt.

CASE 9 Rec_no 303 in Table 6.5 can denote undesired access to Alice's s.txt. The reason can be that Alice has never used the 'dummyAppU' application (rec_no 303 column 3 Table 6.5) to access s.txt though the id in the table is Alice's id. Thus, she may suspect that attackers may have accessed s.txt.

6.3 Discussion of the results for the providers

This section discusses how the results can benefit providers. This includes discussion of how the results from the implementation can assist in mitigating risks associated with threat 1 for the providers.

Assumptions for discussion of the provider side. Section 6.1.2 discussed system architecture of the proposed logging system (Figure 5.3) which deploys F3. Deploying F3 as the history of critical files in disk0 which is controlled by a provider, also leads to integrity or privacy concerns of F3, as discussed in Section 3.3.3. This issue needs to be addressed before deploying F3.

Section 3.4.3 already discussed the possible solution for the issue of F3. Briefly, it could be that a TTP, not a dom0 or provider, should handle our logging system. Thus, the TTP can manage and maintain the logging system (P1 or libVMI, the logging application or app0, and F3) in a special-privileged domU which operates this system.

To maintain integrity of this TTP domU, for instance to prevent the provider from altering or learning about F3, the trusted computing related-research could be the solution. An example is the work in [52] which proposes a solution based on the TPM and offers protection against a malicious provider who has full privilege over a domU in an IaaS.

For discussion in this section, we assume that the history of critical files or F3 is managed by a TTP, not a dom0. Thus, the history files can be trusted even though the dom0 is compromised. When the history of critical file is available to a provider, Subsection 6.3.1 and 6.3.2 below discuss how the results in this implementation assists in mitigating the risks associated with threat 1 to benefit the providers.

6.3.1 Dealing with compromising of the dom0

This subsection discusses how the results can be useful for providers for investigation when the dom0 is compromised by an attacker.

This subsection uses Table 6.2 for discussion of how the history of critical files can be used when the dom0 is compromised because of threat 1 such as when domU attacks dom0. In rec_no 103 in Table 6.2, this record can also be undesired access to Alice's s.txt. The incident in the record may be because of domU attacks upon dom0 (threat 1) and subsequent use of this dom0 to attack Alice's domU, as discussed in Section 5.5.2.

After Alice's domU is compromised, attackers may obtain the alice account which includes the user name and password. Then, the attackers log in to Alice's domU with the alice account, and use 'otherApp' appU to read s.txt. From the table, otherApp appU (rec_no 103 column 3 Table 6.2) is recorded in the table as the history of s.txt. The name of this appU or 'otherApp' that is used by the attackers can be used as a trigger or pinpoint by the providers to conduct further investigation into this incident, and consequently into intruders who attack the providers' cloud infrastructure.

6.3.2 Dealing with spamming

Reducing the number of criminal such as spam domUs should maintain the providers' reputation. This is significant because customers may buy the cloud product based on the providers' reputations [131], as also discussed in Section 5.5. Thus, this subsection discusses how the process behaviour logs help a provider to identify spam activities in his cloud infrastructure.

rec_no	p_nm	a_file	p_ownId	time
1	mail	-	1002	t1

Table 6.6: A process behaviour log file to show the malicious mail command involving spam activities

rec_no	p_nm	a_file	p_ownId	time
1	cat	addr.txt	1002	t1

Table 6.7: A process behaviour log file to show the malicious cat command involving spam activities

Section 3.4.2.1 has already discussed the simulation of spam activities in domU in the case study. This case study simulates spam activities by assuming that spammers rent a Linux VM/domU from an IaaS provider. They then use appU or the *mail* command to send a spam email to a victim.

However, the mail command can be used as *mail -s spamSubj $(cat addr.txt)*. In this case, this mail command sends emails to all victim email addresses in a text file *addr.txt*. Thus, this mail command involves *addr.txt*. Hence, this file could be very important evidence to identify these spam activities. This section discusses how process behaviour log files can be used to assist in identifying spam activities.

When mail and cat commands combine to send spam emails as *mail -s subject $(cat addr.txt)*, Table 6.6 and Table 6.7 can be logs of process behaviour of mail and cat respectively. Table 6.6 shows that the mail command in column 2 is activated by a user with id 1002 in column 4 at time t1 in column 5. Column 3 in this table is blank because the mail command does not access any file, unlike the cat command. Table 6.7 shows that the cat command in column 2 is also activated by the same user with id 1002 (column 4), which activates the mail command and at the same time t1 in column 5. This table also shows that the cat command has accessed addr.txt in column 3.

When both tables above are available to a provider, he can analyse them and may see that this domU owner with id 1002 (column 4 in both tables) uses the combination of both commands to send spam emails: cat to read *addr.txt* (Table 6.7 column 3); then mail (Table 6.6 column 2) to send emails to all the victim addresses in *addr.txt*. Note that, for this discussion, we assume that the information from both tables is sufficient to identify these spam activities. Thus, process behaviour logs from both tables can be useful to assist in analysing and identifying spam domUs in the providers' IaaS cloud. We do not discuss identifying an absolute spam domU. This would involve more research on the identification of spam application behaviour.

This discussion is different from the discussion identifying spam activities in Section 5.6.4. In the latter identification is when a provider first pinpoints the mail command in a suspicion spam domU. The provider then carries out further investigation by recording the behaviour of another command such as cat that works with the mail command to send spam emails. Then, the logs from the recording can be used as evidence to assist the provider in identifying this spam domU.

On the other hand for this discussion in Section 6.3.2, identification occurs when both tables are available to the providers. They may then analyse the combination of the tables with the knowledge of the association among commands such as mail and cat. Thus, the process behaviour logs are flexible in their use by the providers as evidence to assist in analysing criminal activities such as spamming in domU (threat 1) with different approaches such as the approaches discussed in this section and the others in Section 5.6.4.

6.4 Comparisons with Related Work in term of TCB sizes

In this section, the proposed system from the implementation is compared to the systems of related work from Section 2.8 in term of reducing TCB size. The comparison includes monitoring and logging systems for accountability in IaaS.

Our proposed logging system architecture or Figure 5.3 deploys libVMI. LibVMI is based on six high level requirements of programming guidelines or good security guidelines [39]. It is designed to work in the Xen-base environment which is currently virtualisation layer of many cloud providers include EC2 [17], Rackspace Cloud [146] as argued by [43; 59]. Therefore, our architecture inherits these requirements while achieving the history of critical files.

The first two requirements involve a trusted computing base or TCB which is a significant factor when building logging systems in the cloud, as discussed in Section 2.13. A TCB size of a logging system can be compared to a TCB size of another logging system, as discussed in Section 3.5.2.8. The biggest TCB size of a logging system is shown in Figure 3.6. This is when the system's pattern deploys P3 such as a pattern of JAR logging in Figure 4.11. This TCB size includes hw0, hypervisor, dom0, and domU. Figure 3.7 is the smallest TCB size of a logging system. This is when the system's pattern deploys P5 such as a pattern of AVMs in Figure 4.12. In this case this TCB size includes only hw0 and hypervisor.

6.4.1 Comparison with logging work

This subsection compares the logging system in this chapter with those in related work from Section 2.8 based on TCB and achieving a history of critical files.

In this comparison, a logging system is composed of logging processes or Px and log files or Fy. To be simplified, we omit hw0 to be included in a TCB size because all TCB sizes in this thesis include hw0. Our proposed logging system can achieve the history of critical files while yielding a smaller TCB size compared to JAR logging [76], Flogger [30], PASSXen [43]. AVMs system [44] TCB size is smaller than ours. However, this system is not designed to obtain the history of critical files, but may be modified to do so.

Firstly, Flogger and PASSXen can provide a history of critical files. However, Flogger (Figure 4.4) has logging processes that are distributed across domU and dom0. PASSXen (Figure 4.5) also has logging processes that are distributed across the hypervisor, domU, and dom0. Therefore, the proposed architecture (Figure 5.3) yields less TCB than Flogger and PASSXen. This is because the TCB of our system includes only hypervisor and dom0, not domU, whereas both their TCB include domU.

Secondly, a TCB size of AVMs (Figure 4.12) includes only hw0 and hypervisor (VMware), this is smaller than one of the proposed architecture in this thesis. However, this system is to record incoming and outgoing network packets of domU (to detect cheating in on-line gaming) not to record the history of the critical file, but could be modified to do so.

The last work is JAR logging (Figure 4.11). To apply this approach in an IaaS environment, when a user's JAR file is stored in diskU and to activate the logging mechanism in this file, domU needs JVM. To add JVM in domUs increases the TCB.

Table 6.8 concludes the comparison between the proposed logging system and

Criteria	Our system (Figure 5.3, 2013)	JAR logging [76] (Figure 4.11, 2012)	Flogger [30] (Figure 4.4, 2011)	PASSXen [43] (Figure 4.5, 2011)	AVMs [44] (Figure 4.12, 2010)
Achieving the history of critical files	YES	YES	YES	YES	NO
TCB	YES (hypervisor, dom0)	NO/larger than our approach (hypervisor, dom0, domU)	NO/larger than our approach (hypervisor, dom0, domU)	NO/larger than our approach (hypervisor, dom0, domU)	YES/ smaller than our system (hypervisor)

Table 6.8: TCB Comparisons between the proposed logging system and related logging works

the systems of the related logging work. From the table, our system can achieve the history of critical files while yielding a smaller TCB size compared to JAR logging, Flogger, and PASSXen.

This is because the TCB size of our system includes only two components: a hypervisor and dom0; whereas the others include three components: hypervisor, dom0, and domU. A TCB size of AVMs includes only a hypervisor; whereas ours includes a hypervisor and dom0. Thus, AVMs' TCB size is smaller than ours. However, AVMs is not designed to achieve the history of critical files.

6.4.2 Comparison with monitoring work

In this subsection, the TCB size of the proposed system is compared to the TCB sizes of the monitoring systems from previous works

As discussed in Section 2.9, monitoring work does not use logging file or Fy. However, these works below could be modified to perform logging tasks. All the works also deploy XenAccess or libVMI. The TCB sizes of a network monitoring application in [1] and a demo monitoring program in [39] is the same as the proposed logging system's TCB size. However, only the proposed logging system achieves the history of critical files while both of them do not, but could be modified to do so. Lares [77] does not achieve both the history of critical files and TCB size.

Firstly, Lares [77] can monitor domU behaviours. It may be possible to modify this system to collect the history of critical files. However, this requires insertion of hooks at runtime into a domU, which increases TCB. Secondly, [1] propose a network monitoring application that identifies which process inside a Windows domU is responsible for malicious network traffic leaving this domU.

Lastly, [39] present a demo monitoring program in dom0 that outputs all file/directory creation/removals process which are happening in domU's /root directory. Both works deploy XenAccess in dom0. Thus, these systems have small TCB (hw0, Xen, and dom0) the same as the proposed architecture in this thesis. However, they are not designed to achieve a history of critical files, but could be modified to add this functionality.

Table 6.9 concludes the comparison between the proposed logging system and

Criteria	Our system (Figure 5.3, 2013)	A network monitoring application in [1] (Figure 4.13, 2011)	Lares [77] (Figure 4.10, 2008)	A demo monitoring program in [39] (Figure 4.13, 2007)
Achieving history of critical file	YES	NO	NO	NO
TCB	YES/(hypervisor, dom0)	YES/the same as our approach (hypervisor, dom0)	NO/larger than our approach (hypervisor, dom0, domU)	YES/the same as our approach (hypervisor, dom0)

Table 6.9: A Comparison between the proposed logging system and related monitoring works

related monitoring works. From the table, the TCB sizes of a network monitoring application in [1] and a demo monitoring program in [39] are the same as the proposed logging system's TCB size. However, only the proposed logging system achieves the history of critical files, while both of them do not, but could be modified to do so. Lares [77] do not achieve both history of critical files and TCB size.

6.5 Performance measurement of the proposed logging system

This section provides the performance measurement of our proposed logging system (Figure 5.3) in term of its accuracy in capturing logging information (e.g., a file name of a file that is read by a domU's application) from a target virtual machine or domU. **The result is that the proposed system has 100% accuracy in capturing log information when an application or appU in a target domU accesses a file for at least 65 milliseconds**.

The lists below discuss the aim of the performance measurement of the proposed logging system in terms of accuracy in this section, and the experiment sets to calculate the accuracy.

- **Aim of the performance measurement of the proposed logging system in term of accuracy**

  The aim of the measurement is to guarantee the proposed logging system will yield 100% accuracy in capturing the information from memU in domU; how many milliseconds the read application or appU needs to be in memU after finishing reading a file such as s.txt, and before closing the file. The answer can be interpreted as that the proposed system in dom0 yields 100% accuracy in capturing the log information if an application in domU accesses a file for at least x milliseconds.

- **Experiment sets to calculate the accuracy**

  One set of experiments is as follows. (i) We set a particular sleeping time such as 80 milliseconds for a read application after it finishes reading a file and before it closes the file. (ii) The read application will be run 1000 times to perform Step (i). Then, (iii), the proposed system needs to capture log information (a file name string of a file that is read by read application) from all 1000 runs of the read application.

During a particular run of the read application to read s.txt, if the proposed system captures (from memU) the correct file name or the string 's.txt' that has been read by this application in this particular run, this capture is called a 'hit'. When a read application is run for 1000 times and the proposed system needs to capture all these 1000 times, then we can count a number of hits between 0 and 1000.

For each experiment set or steps i to iii, the accuracy (in percentage) is calculated by the number of hits divided by 100. Thus, the accuracy is ranged from 0% to 100%. We run each experiment set for 10 times, then calculate the average accuracy of this experiment set. Thus, the average accuracy is calculated by the summation accuracy of each experiment set (in percentage) from the first time to the tenth time divided by 10.

The remainder of this section is as follows. Subsection 6.5.1 discusses the concerns in terms of performance measurement of logging systems in the cloud. Subsection 6.5.2 explains our performance measurement environment of the proposed logging system. Then, Subsection 6.5.3 describes the experiment design (in both domU and dom0) for our performance measurement of the proposed logging system. Subsection 6.5.4 discusses what it means when the logger can capture the correct and incorrect information from memU. How to run the experiment to collect the accuracy of the logger is explained in Subsection 6.5.5. Then, Subsection 6.5.6 reveals and discusses the results for the accuracy of the proposed logging system. Finally, Subsection 6.5.7 provides a summary.

6.5.1 Performance concerns of logging systems in the cloud

This subsection discusses what the concerns in term of performance measurement of logging systems in the cloud can be from the literature. The concerns can be considered from both domU and dom0.

To measure the performance of logging systems, we can consider two main performance concerns of logging systems in the cloud: performance concerns of

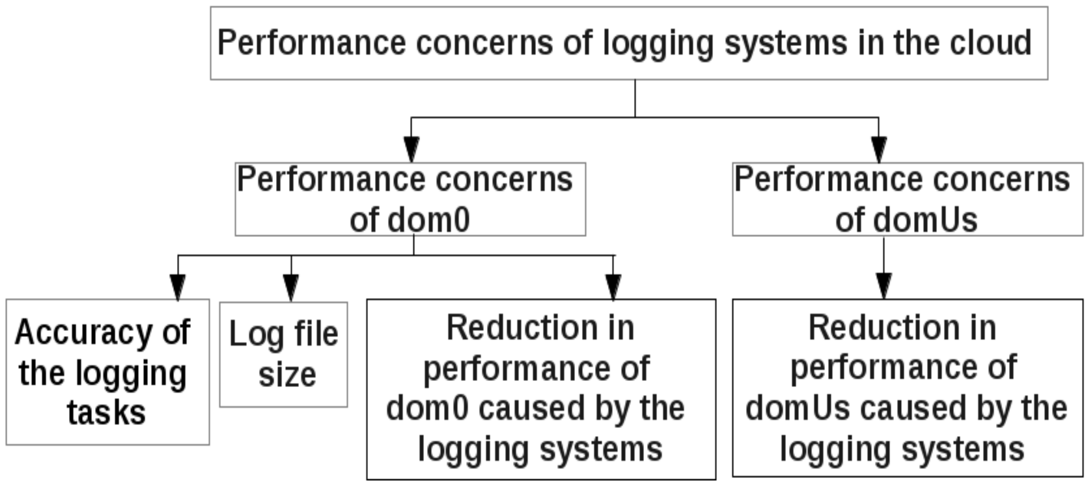

Performance concerns of logging systems in the cloud
Performance concerns of dom0
Performance concerns of domUs
Accuracy of the logging tasks
Log file size
Reduction in performance of dom0 caused by the logging systems
Reduction in performance of domUs caused by the logging systems

Figure 6.5: Performance Concerns of logging system in the cloud

dom0; and performance concerns of domUs, see Figure 6.5. The performance concerns of dom0 include accuracy of the logging tasks, log file size, and, reduction in performance of dom0 caused by the logging systems, full details are given in Section 6.5.1.1. The performance concerns of domUs are the reduction in performance of domUs caused by the logging systems, more details are given in Section 6.5.1.2. Both concerns can be applied to interception and introspection logging approaches. Here, we consider the measurements based on the introspection approach which is deployed in the proposed logging system for this thesis.

6.5.1.1 Performance concerns of dom0

This section discusses the performance concerns of dom0 which deploys logging systems. We discuss three of the performance concerns: accuracy of the logging tasks, log file size, and reduction in performance of dom0 caused by the logging systems. However, we experiment on only the measurement of the accuracy of the logging tasks.

1. **Accuracy of the logging tasks**

   We discuss the performance measurement only for the introspection approach for the logging system. This is because this approach needs to access the main memory of domU, which is volatile. The accuracy of the logging tasks in accessing the volatile memory is the first consideration of the performance measurement. Therefore, this chapter presents only the measurement of the accuracy. Section 6.5.2, 6.5.3, 6.5.5 and, 6.5.6 provide details of the accuracy of our proposed logging system, how we measure it, and the measurement results and discussion.

2. **Log file size**

   [30] state that file-centric logs can grow at a relatively higher rate compared to system-centric logs. They also argue that tiered storage and archival, de-duplication and summarisation techniques may be solutions. [44] also states that present hard disk capacities are measured in terabytes; thus, this should be a solution to the log file size explosion problem.

3. **Reduction in performance of dom0 caused by the logging systems**

   The performance impact can be measured as a reduction in performance of dom0 caused by the monitoring software. For example, this measurement can be achieved by measuring how much CPU of a machine is consumed by the logger process to achieve logging tasks, compared to the machine that does not deploy the logger process. This measurement should be done when the full implementation of a logging system is ready, or after the accuracy of the logging processes in a particular logging system satisfies the accuracy requirement of the system.

6.5.1.2 Performance concerns of domUs

For logging systems that deploy the introspection approach, the systems need to access the main memory of domUs. This may lead to a reduction in performance of domUs caused by the logging systems. Our proposed logging system reuses libVMI [79] previously known as the XenAccess [147] tool. This tool imposes a minimal performance overhead to the target domU memory [39].

6.5.2 Performance measurement environment

This subsection describes the experimental environment of this performance measurement. This includes application and hardware components.

- **Application components of the Experiment**

  We use the same set of experiments in the implementation of the proposed logging systems as in Section 6.1. A different machine is used in this measurement testing.

  In this experiment, libVMI can obtain the names and IDs of all running processes in the processes list (Figure 5.5) by accessing the memU of a target domU. Then, the logger repetitively calls libVMI in a loop to check whether the list contains a process name called 'read' or not. Section 5.6.3.1 has already discussed how a process, command, or appU such as read appU works in the Linux system.

  In this measurement setting, it assumes that Alice runs read appU 1000 times, and each time the application reads s.txt; thus, ideally, the goal of the logger is that it has to capture the information of these 1000 times of read appU activities. Then, it stores the captured information in log files.

- **Hardware**

  The experimental environment comprised a single physical machine as hw0. Its hardware configuration includes an Intel Core 2 Quad CPU Q9400 @ 2.66GHz x 4 (64-bit) CPU and 2.7 GiB of main memory. Hw0 consisted of hypervisor that used Xen version 4.1.4 (preserve-AD) with a Fedora 16 dom0 running a 64-bit Linux kernel, version 3.6.11-4. Alice's domU is running a 64-bit Linux kernel version 3.6.11-4 for Fedora 16. It consisted of 989.7 MiB of main memory. Networking of both dom0 and domU is disconnected.

6.5.3 Experiment design

This section explains the experiment design for both domU and dom0.

6.5.3.1 Experiment design for domU

These lists below explain the experiment design for domU.

- **Setting up**

  Section 6.1.1 has already discussed the context of a domU in the implementation and the aim of the proposed logging system. DomU context and the aim of the logger in this measurement testing is exactly the same as in Section 6.1.1. Thus, Figure 5.3 shows the context of a domU for this measurement experiment. Figure 6.6 illustrates the experiment set-up for the accuracy measurement.

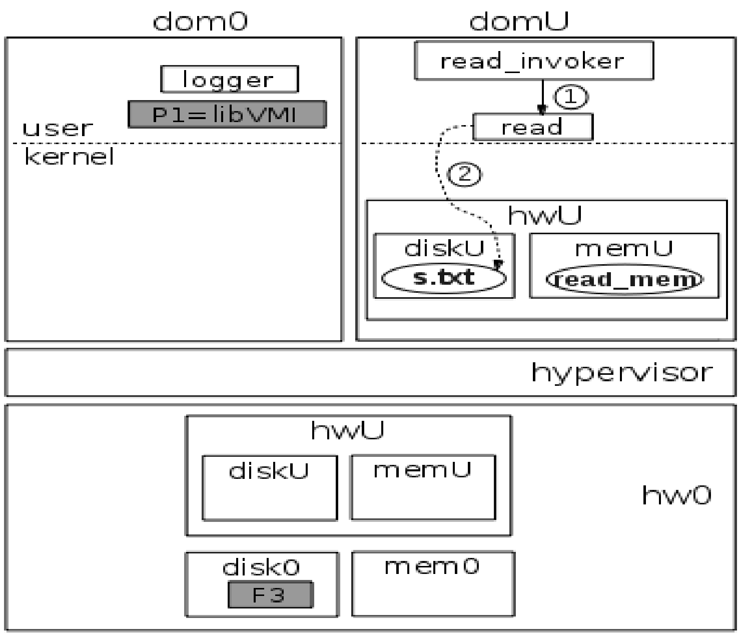


Figure 6.6: The experiment set-up to collect the accuracy measurement

- **Read application routines**

  From Figure 5.3, read application routines are as following. Note that, a text file or s.txt that the read application accesses contains only a simple line of text as "9, 9".

  Read application routines are to:

1. open s.txt file,
2. read the file and print the file's contents to a screen,
3. close the file,
4. be terminated.

- **Read_invoker**

  After the read application is started (see Step 1 in Figure 6.6) by the read_invoker application (discussed below), the read application will then perform (see Step 2 in Figure 6.6) the routines described above. Read application will be run by an application called read_invoker. The invoker will run read for 1000 times, repetitively performing steps 1 and 2.

- **Read application routines with suspension**

  For the measurement experiment, read application routines are modified to add suspending execution for microsecond intervals as will be shown below. The suspension is performed by C programming *usleep* function [148]. The time intervals range from 0 to 100 milliseconds. Thus the read application will sleep after finishing reading s.txt. Then, it wakes up, closes the reading file, and is terminated. Thus, the routines below are one run of a read application.

  However, read will be run by another application called read_invoker. The invoker will run read for 1000 times. Note that, the average processing time of read application in the setting domU (without suspending sleeping or when setting the sleeping time to 0) is about 400 microseconds. We obtained this time number by running read application for 100,000 times.

  Read application routines with suspension are to:

  1. open s.txt file,
  2. read the file and print the file's contents to the screen,
  3. **suspend or sleep for x milliseconds (x = 1000)**,
  4. close the file,
  5. be terminated.

- **Running read application for the experiment**

  To run read application, all 1000 rounds of the read application run will be performed by read_invoker which will be started only once. The experiment set-up for read application to be run for 1000 times in a domU is now ready. Section 6.5.3.2 discusses how to design the logger to capture the log information from all read application runs.

6.5.3.2 Experiment design for dom0

The lists below discuss how to design the logger in dom0 to capture the log information from all read application runs.

- **Setting up.** The logger in this experiment has the same system architecture as the proposed logging system (Figure 5.3) in Section 6.1.2. Thus, it has exactly the same mechanisms as the one in Figure 5.3. Subsection 6.1.3 explains the routines of the logger in the propose system architecture. Figure 6.6 provides an overview for both the logger and read application for this experiment. In this figure, the main components of the proposed logging system in this experiment are logger, P1 or libVMI [79], and F3 as a log file.

  Again, libVMI is a C library that can read the memory space or *read_mem* in memU from domU, see Figure 6.6. The logger finds a read process in the memU of the target domU by checking (using a loop) from the beginning to the end of the linked list of all running processes (Figure 5.5) in the domU.

- **The logger for this experiment.** For this measurement, we modified the logger in Figure 5.3 to be able to capture the log information of each run of read application that will be run for 1000 times. Thus, the logger in Figure 6.6 is capable of doing so. **The pseudocode of the modified logger is presented below.**

```
1. Set totalReadRounds = 0, hit = 0
```

```
2. Check if read application is in the linked list
   if (read application is in the list)
      2.1 Check if the application has read s.txt
         if (the application has read s.txt)
            if (the capturing file name is the string s.txt)
               hit++
         else
            goto 2.1
   else
      goto 2
3. totalReadRounds++
4. if (totalReadRounds = 1000)
      goto 5
   else
      4.1 Check if a brand new read application exists in the list
         if (a brand new read application is existed)
            goto 2.1
         else
            goto 4.1
5. end
```

- **Running the logger**

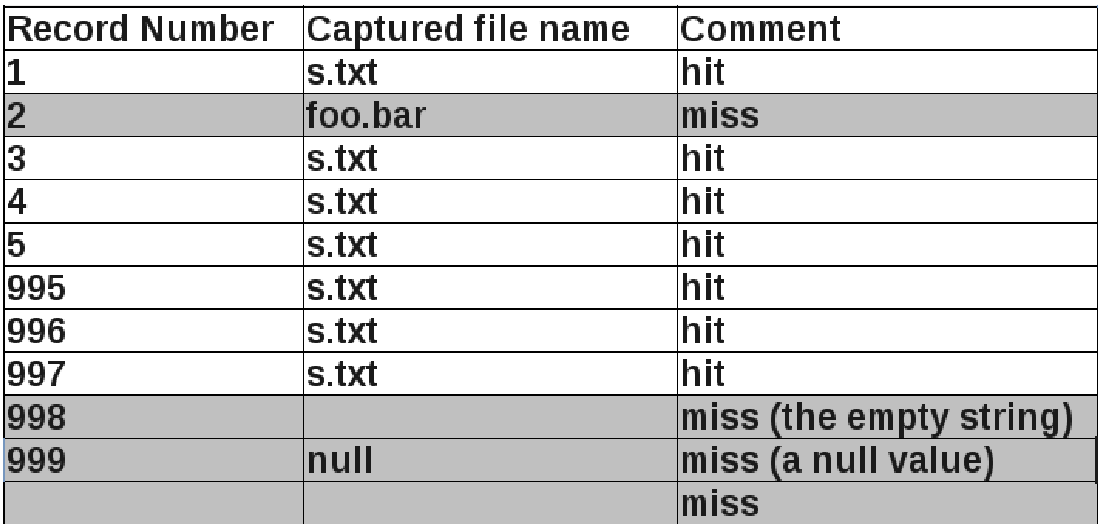
  The logger in dom0 is run before read_invoker in domU. It is run only once to capture the log information from each one of all 1000 read application runs.

6.5.4 Collecting hits and misses

This subsection discusses how the logger can capture the correct information from memU. Ideally, the logger must capture information from memU for all 1000 read application runs. Each run has reading s.txt as in the routine shown above.

6.5.4.1 Hits

We call the correct capture a hit when the logger can capture the correct information or the string "s.txt". Figure 6.7 is an example of the output spreadsheet files in this experiment. From the figure, record number 1, the third column as comment, is a hit because the captured information (the second column as Captured file name, in Figure 6.7) is 's.txt'. The record number 6 to 994 is omitted.



Record Number	Captured file name	Comment
1	s.txt	hit
2	foo.bar	miss
3	s.txt	hit
4	s.txt	hit
5	s.txt	hit
995	s.txt	hit
996	s.txt	hit
997	s.txt	hit
998		miss (the empty string)
999	null	miss (a null value)
		miss

Figure 6.7: Hits and Misses

They are assumed to be the same as record number 1; thus, each of them is also a hit. Thus, the overall hits are in record number 1 (1 hit) and record number 3 to 997 (995 hits). In total, the number of hits is 996 (1 hit plus 995 hits), which is 99.96% of the accuracy in this case.

6.5.4.2 Misses

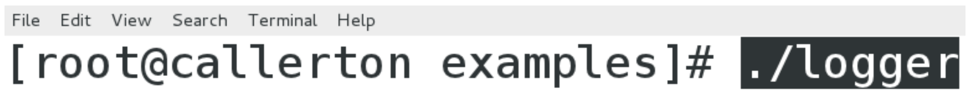
Miss is when the logger captures incorrect information. This can be another string instead of "s.txt" (record number 2, "foo.bar" instead of "s.txt"), an empty string (record number 998), or a null value (record number 999).

In this measurement testing, we also consider a miss as when the logger cannot capture the read appU in memU. For example, when the read application is run for 1000 times, but the logger can capture the runs only 999 times. For example, the last row in Figure 6.7 expresses when the logger cannot capture the 1000th run of the read application. Thus the last row is one miss. It is unacceptable if the logger cannot capture the 1000 running times. This is because the captured information of any run of the read application can be very important evidence that may be used in law courts. Thus, from the figure, the number of overall misses is four (record numbers 2, 998, 999, and the last row). All accuracy results of the proposed logging system will be presented and discussed in Section 6.5.6.

6.5.5 Running the experiment to collect accuracy of the logger

This section explains how to run the experiment to collect the accuracy of the logger.

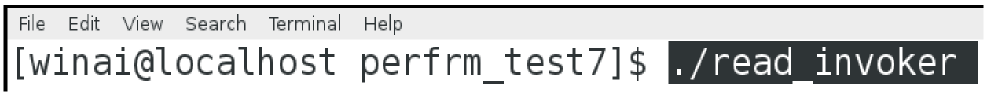
Section 6.5.3 has already discussed the design and set-up of the logger, and read and read_invoker applications to perform the accuracy measurement. Section 6.5.4 then explains how to collect hits and misses. To perform the experiment, the logger (Figure 6.8, the black highlight) will be started before read_invoker (Figure 6.9, the black highlight), and this is one experiment set. To



```
[root@callerton examples]# ./logger
```

Figure 6.8: To run the logger in dom0

collect the average accuracy of a particular sleep time such as 100 milliseconds, we run each experiment set for ten times for each sleeping time setting, then the



```
[winai@localhost perfrm_test7]$ ./read_invoker
```

Figure 6.9: To run the read_invoker application in domU

average accuracy from these ten times is calculated and collected. The sleeping time intervals are set to vary from 100 to 0 to find the least time (in milliseconds) that the read application needs to sleep for after finishing reading a file and before closing the file. This will be discussed in Section 6.5.6, along with the results.

6.5.6 Results for accuracy of the proposed logging system

This section reveals the results of the accuracy of our proposed logging system, and presents a discussion of the results. Then, it demonstrates the decreasing trend of the accuracy of the logger, and the resulting discussion.

6.5.6.1 The accuracy is 100% when the read application accesses a file for at least 65 milliseconds

The results show that the optimum sleep time is 65 milliseconds for the read application. This implies that the accuracy of the proposed logging system is 100% when any application in domU accesses (opening a file until closing the file) a file for at least 65 milliseconds. Figure 6.10 is the graph of the result. From the graph, when the sleeping times are from 80 to 65 milliseconds, the accuracy is 100%.

However, the first time that the accuracy is less than 100% (99.98%) is when the sleeping time is 64 milliseconds. Thus, the minimum least sleeping time for the accuracy to be 100% is 65 milliseconds, see the dotted line. The accuracy decreases as the sleeping time moves from 64 until 59 milliseconds. However, the accuracy in this interval is only marginally different. It decreases from 99.99%, to 99.95%. The next graph presents the trend of the decreasing accuracy when the sleeping time is set from 100 to 0 milliseconds with the reduction of 10 milliseconds each time, in Section 6.5.6.2.

To improve the accuracy, the logger process can be run in dedicated CPUs. Ideally, if we double the CPU number, the sleeping time needed to get 100% accuracy should be decreased by half such as from 65 milliseconds to 32.5 milliseconds. However, we did not test this doubling approach.

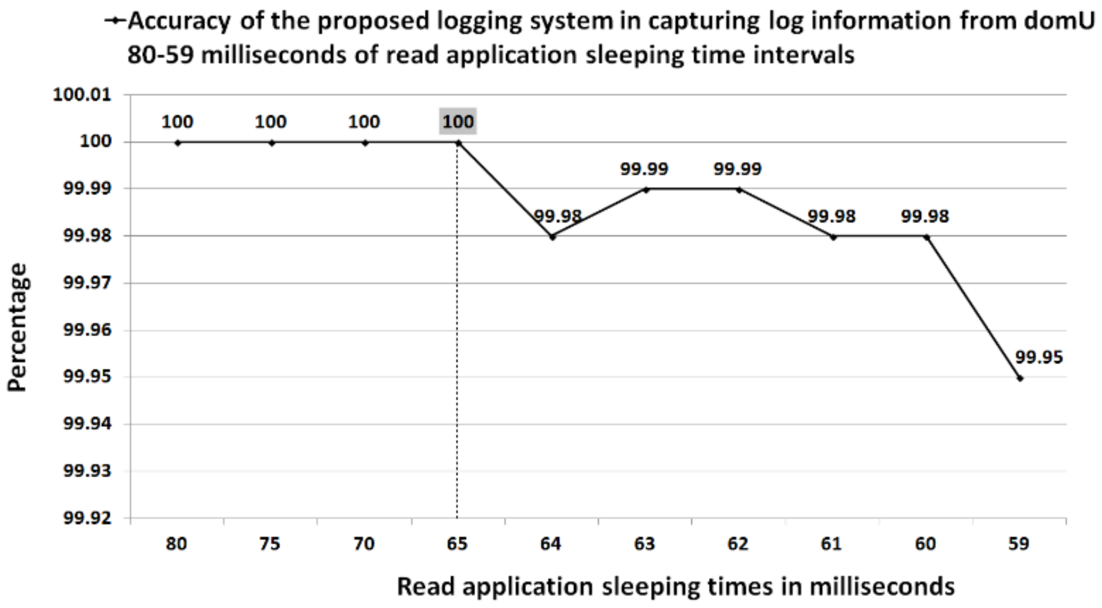


Figure 6.10: A graph showing the accuracy of the proposed logging system in capturing log information from domU

6.5.6.2 Decreasing trend of the accuracy of the logger

In order to present the decreasing trend of the accuracy, we set up the sleeping time intervals from 100 to 0 milliseconds with the reduction of 10 milliseconds for each experiment set. The second graph (Figure 6.11) presents the decreasing trend of the accuracy for the proposed logging system in capturing log information from domU. From the graph, the logger yields 100% of the accuracy when the read application sleeping time is from 100 to 70 milliseconds. The accuracy is 99.98%, 99.99%, and, 93.98% when the sleeping times are 60, 50, and 40 milliseconds respectively.

The accuracy sharply decreases from 93.98% to 31.75% (see the dotted lines) when the sleeping time is changed from 40 to 30 milliseconds. From the experi-

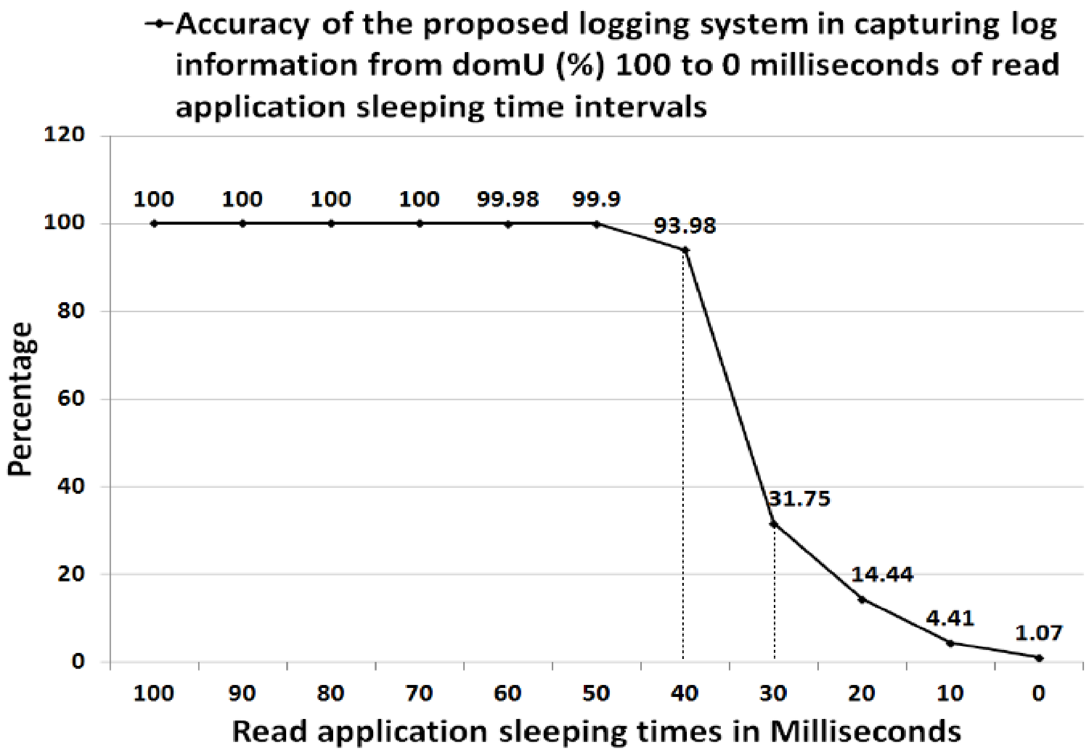


Figure 6.11: A graph showing the accuracy of the proposed logging system in capturing log information from domU

ment, starting from the 30 milliseconds until 0 millisecond, the logger seems to be halted. This may be because of the loop in the logger, used to traverse the processes list (Figure 5.5), is not fast enough. Thus, it should be possible to increase the speed of the loop by running the logger on dedicated CPUs in a multi-core system. However, this issue is out of the scope of this thesis, and needs further investigation for clarification. Note that, when the sleeping time is 0, the read application runs without suspension for experiment. It is an actual running of the read application.

6.5.7 Summary

This section provides the performance measurement of our proposed logging system in terms of its accuracy in capturing logging information from a target virtual machine or domU. The system has 100% accuracy when an application in a target domU accesses a file for at least 65 milliseconds. To improve the accuracy, the logger process can be run via dedicated CPUs.

A benchmark of accessing times to access a file can be how many milliseconds that an application spends to access, open, read, write, or close its database file. This benchmark can be difficult to measure. This may be because the benchmark measurement can be heavily related to hardware specific of domUs. A hardware specific qualities can be different from domU to domU. This specification can be: storage (e.g., a solid state drive or SSD, or hard disk drive or HDD) of a file such as s.txt; the amount of main memory; the system load; size of the file; the file system in use (e.g. Linux third extended file system or ext3, or Microsoft Windows NTFS), CPU speed; and so on. We did not find such benchmark to compare with the minimum sleeping time for the measurement result which is 65 milliseconds.

Thus, we cannot compare the results to a benchmark of the average time taken for an application to access a file. However, the results can be a basis to clarify the ability of the logger in capturing the log information. This clarification may be used as a guideline to efficiently and appropriately design, implement, and deploy logging systems. As a result, this can truly enable the logging systems to work in real world cloud production systems.

6.6 Conclusions

This chapter has addressed three research gaps. The first one or research Gap 1 (b) is the lack of concern for reducing the trusted computing base or TCB size of a logging system. The second one or research Gap 2 is an issue that research relating to logging in IaaS only focuses on system-centric logs which is the hardware layer log, such as memory use, disk storage, temperature, and voltage. The third research gap or research Gap 3 is a lack of analysis of what the contents of the log file should actually be, and of how the contents can be used to deal with the real world CSA threats to benefit both sides in detail.

The chapter has three main contributions. Firstly, the design of our proposed logging system yields TCB size compared to previous work. This contribution addresses Gap 1 (b). All logging related components (an introspection tool that we have re-used and a logger application) that have been deployed in building the proposed logging system in this chapter are inside dom0.

The TCB size of a logging system can be compared to a TCB size of another logging system, as discussed in Section 3.5.2.8. For example, a TCB size of a logging system, which includes all components (hw, a hypervisor, dom0, and domU) is bigger than a TCB size of another logging system, which includes only hw, a hypervisor, and dom0. The TCB size of our architecture includes only a hypervisor and dom0, not domU. In contrast, previous works have placed some of their logging-related components in domU; thus, the TCBs of their systems have to include domU in addition to a hypervisor and dom0. Other previous work that have deployed the same introspection tool as in this chapter yields the same TCB as ours. However, they are not designed to log the history of critical files.

Secondly, our system collect file-centric logs rather than system-centric logs. This contribution addresses research Gap 2. The prototype implementation of the proposed logging system can be an alternative to collect file-centric logs to enhance accountability in IaaS by domU's VM memory introspection approach (by traversing the domU kernel file structures from dom0).

Moreover, the file-centric history logs can be associated with a process and files in a domU, for example, a record of a process P which reads file F. The proposed log files differ from previous work which focus only on system-centric logs (e.g., the connection topology, bus speeds, and processor loads).

Lastly, this chapter has presented how the results from the proposed logging system can assist in mitigating the risks associated with threat 1 with nine incident scenarios. This contribution addresses research Gap 3. These scenarios illustrate how the results from the implementation can be used to form the history of critical files which can be used to assist in mitigating risks associated with CSA threat 1 for a customer when the customer is the only user in the domU or if she shares the domU with many users. We then discussed how the results can help to form process behaviour log files to assist in mitigating the risks associated with threat 1 to benefit providers. This discussion included how the process logs can be used to investigate the causes when a dom0 is compromised, and when attackers use domUs for spamming activities.

Thus, with systematic approaches, this shows that our proposed solutions can assist in mitigating the risks associated with real world IaaS issues and many scenarios. The proposed solutions also benefit both customers and providers.

The next chapter briefly summarises the contributions of this thesis and describes some threads of potential future work.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Infrastructure as a service (IaaS) is a base for other types of cloud, and is increasingly being used by individuals and organisations. Accountability that consisting of logging systems as a core is necessary to build trust.

However, previous work that relate to logging/accountability for IaaS has the following research gaps. **The first research gap or Gap 1** is a lack of systematic approaches to building logging systems in IaaS. This includes the lack of simultaneous consideration of both customers and providers or Gap 1(a); the lack of concerns for reducing the trusted computing base or TCB size of a logging system or Gap 1(b); and the lack of security analysis of the logging systems themselves before the deployment of the systems or Gap 1(c).

The second research gap or Gap 2 is that work that relates to logging in IaaS only focuses on system-centric logs (which are usually disclosed to consumers e.g., disk storage or temperature) rather than file-centric logs such as tracing customers' files in VMs from the time they are created to the time they are deleted. **The third research gap Gap 3** is the lack of analysis of what the contents of the log file should actually be, and how the contents can be used to deal with the real world CSA threats to benefit both sides in detail. **The last research gap or Gap 4** is that lack of descriptions of logging systems in the context of a design pattern of the systems' components.

The aim of this thesis has been **to provide support for accountability in the cloud with systematic approaches to mitigate the risks associated with real world CSA threats, to benefit both customers and providers**. To achieve this aim, the project has satisfied the objectives listed below.

- Objective 1: understand the real world cloud problems (the CSA threats), and accountability with logging approaches in the IaaS
- Objective 2: design a generic framework of logging solutions to mitigate the risks associated with the CSA threats
- Objective 3: define, identify, and draw conclusions on the advantages and disadvantages of logging system patterns, then analyses existing works in relation to the patterns
- Objective 4: design and implement logging systems based on the generic framework from Objective 2 and the identified patterns from Objective 3 to mitigate risks associated with a specific threat (threat 1 which is *abuse and nefarious use of cloud computing*, for example when cloud customers rent VMs then use these VMs to conduct criminal activities), and to the benefit of both cloud customers and providers
- Objective 5: evaluate the proposed generic framework from Objective 2, and the proposed logging systems from Objective 4

  We evaluate the framework by demonstrating instantiations of logging solutions to deal with CSA threat 1 in Chapter 3 (the spamming case study) and in Chapter 6 (a proposed logging system to benefit both customers and providers). We then evaluated the performance of the proposed logging system in terms of the accuracy of the systems in Chapter 6.

The remainder of this chapter is structured as follows. Section 7.1.1 provides a summary of the contributions of this thesis. Finally, future work relating to this thesis is discussed in Section 7.2.

7.1.1 Summary Of Contributions

This thesis provides a number of key contributions to address the research gaps, and to satisfy the aim and the objectives stated above:

1. **An in-depth background and literature review of CSA threats and systematic support for accountability**

   This contribution addresses research Gap 1. This background and literature review summarises the key concepts of IaaS, CSA threats to IaaS, accountability/logging approaches to mitigate the risks and the problems of the approaches, and systematic support for accountability in the cloud.

   This summary differs from previous work which focus on dealing with cloud problems without full consideration of the main involved/systematic aspects necessary to provide logging systems. For example, previous works do not fully discuss the security analysis of logging systems themselves before deploying them into the real world IaaS productions. Without adequate consideration of the systematic aspects, it is difficult to efficiently and effectively enable logging systems.

   The value of the systematic approach is to provide a clear vision of the logging system's development in the cloud, such as the security analysis of the systems. Subsequently, the systematic approach can efficiently and effectively enable accountability in the cloud. Accountability in the cloud is an important concept to assist in mitigating the risks associated with CSA threats.

2. **Generic logging components for IaaS cloud**

   This contribution addresses research Gap 1 (c). To facilitate systematic support for accountability in the cloud, these generic logging components provide ways to build logging systems. The value of the generic logging components are to encompass all possible instantiations of logging solutions for IaaS cloud, and to provide a clear view of all components that relate to logging systems in IaaS. This view provides a basis for the analysis of the logging systems' security before their deployment.

Thus, the generic logging components enable logging systems to be appropriately designed or manipulated by participating cloud parties (a provider, customer, or auditor). As as result, this enhances systematic support for accountability in the could.

3. **Analysis of how CSA threat 1 affects both customers and providers simultaneously; and the proposed logging solutions to assist in mitigating the risks associated with the threat for both**

   This contribution addresses research Gap 1 (a). The analysis illustrates how CSA threat 1 such as mis-usage of customer VMs can affect both sides. This analysis differs from previous work which usually concern the effects of the threats to only either the customer or provider, providing solutions for either side, but not for both.

   The value of the combined analysis is to provide a basis to understand what contents logging solutions need to collect to be used as evidence to deal with threat 1 for both customers and providers.

4. **The design of our proposed logging system yields a smaller trusted computing base or TCB size compared to previous work**

   This contribution is that the size of the TCB for the design of our proposed logging system is smaller than TCB sizes of the proposed logging systems of previous works. This contribution addresses research Gap 1 (b).

   All logging related components (such as an introspection tool and a logger application) which were deployed in building the proposed logging system are inside dom0. Thus, the proposed system in our prototype implementation yields a smaller TCB while obtaining the history of critical files. This is because the TCB of our architecture includes only a hypervisor and dom0, not a customer VM. In contrast, previous works have placed some of their logging related components in the customer VM. Other previous work that deploy the same introspection tool as we have re-used yield the same TCB as ours. However, they are not designed to obtain the history of critical files.

5. **Collecting file-centric logs rather than system-centric logs**

   This contribution is that our proposed logging solutions focus on collecting file-centric logs rather than system-centric logs. This is because research Gap 2 states that current logging research in the cloud focuses on system-centric logs rather than file-centric logs.

   There are some logging solutions that emphasise file-centric logs with an interception approach. However, the prototype implementation of the proposed logging system can be an alternative approach to collect file-centric logs to enhance accountability in IaaS. This approach is the introspection of the customer VM's memory from dom0. The introspection traverses the kernel data structures in the memory.

   The prototype implementation of the proposed logging solutions can collect file-centric log history of customers' critical files. The history information is composed of file-centric logs rather than system-centric logs. The file-centric logs can present associations between a process and files in a customer VM, for example, a record of a process P which reads file F. The proposed log files differ from previous work which focus only on system-centric logs such as the connection topology, bus speeds, and processor loads.

6. **Presentation of how the results from the proposed logging system assist in mitigating the risks associated with CSA threat 1**

   This contribution is that we provide many scenarios to present how the results from the prototype implementation can be used to form log files to assist in mitigating the risks associated with CSA threat 1 to benefit both customers and providers. This contribution addresses research Gap 3.

   To assist in mitigating the risks associated with threat 1 to benefit customers, we discuss the formation of the history of critical files from the results of the prototype implementation. This includes analysing nine scenarios of malicious incidents in a customer VM when, for example, a customer shares her VM with other users.

   To assist in mitigating the risks associated with threat 1 (e.g., when criminals use VMs to conduct spamming activities) to benefit providers, we discuss formation of the process behaviour log files. This shows that our proposed solutions can assist in mitigating the risks associated with real world IaaS issues and many scenarios, and to benefit both customers and providers.

7. **Three proposed design patterns in the context of logging in IaaS cloud**

   This contribution addresses research Gap 4. The proposed patterns facilitate analysis of logging systems in terms of their quality. These patterns could increase reusability of the design and development of logging systems. Designers should access these patterns more easily. The patterns could assist a designer adopts design approaches which make a logging system reusable and not to choose approaches which do not concern reusability concepts. They can enhance the documentation and maintenance of existing logging systems.

   We provide a spectrum of patterns for describing how to construct logging systems with varying characteristics. For developers when building a logging system, the knowledge of characteristics of this system could assist them to get the right design of the system with minimal effort and time commitments. We also clarify why a number of patterns and logging system architectures based on these patterns are missing. To the best of our knowledge, these three logging patterns are not yet described in the literature.

7.2 Future work

The research work presented in this thesis provides a basis for a number of potential related future works as listed below:

1. **The accountability within logging system approaches in this dissertation enables systematic support for accountability to assist in mitigating risks associated with threat 1 for PaaS for both customers and providers**

PaaS provides a solution stack and computing platform to assist in preparation of applications with the low cost and simplicity of purchasing and managing the fundamental hardware and software, and provisioning hosting capabilities [22]. This includes supporting all processes of development and deployment of web applications and services fully available from the Internet [22]. An example of PaaS is the Google App Engine which can also be considered as a web application hosting service [149]. Although this thesis focuses on dealing with the CSA threats to IaaS, these threats present issues for the security of PaaS as well [23].

The layers of PaaS infrastructure are more complicated than the layers of IaaS infrastructure, as agreed by [22; 46; 91] and discussed in Section 3.5.2.1. However, the accountability within logging system approaches in this dissertation should allow the combat of the threats for PaaS. To apply the accountability approaches to mitigate the risks associated with CSA threats for PaaS, as we do for IaaS in this thesis, may be achieved via the following two steps:

- Step 1: Creating generic logging components for PaaS

  PaaS can always be built up by adding extra layers on top of IaaS layers as discussed in the beginning of Section 2.2. Therefore, comprehension of our proposed generic logging components and of logging systems in IaaS could also assist in mitigating the risks associated with the CSA threats applicable to the security of PaaS.

  For example, in [22], integration and middleware layer can be added on top of IaaS to build up PaaS; thus, it is possible that ones can add extra layers on top of our proposed generic logging components to create such components for PaaS.

  We could then considerer the possible and appropriate locations of logging processes or Px and log files or Fy in these new PaaS generic logging components. These locations of Px and Fy in PaaS architecture could affect their security concerns similar as we discussed (in Section 3.5.2.1) for our generic logging components for IaaS in this thesis. Thus, these new generic logging components can be a tool to enable systematic support for accountability in the PaaS cloud.

- Step 2: Applying the history of customer's critical files to assist in mitigating the risks associated with threat 1 in PaaS to benefit both sides

  Following development of the new generic logging components (discussed in Step 1 above) for PaaS as a tool to enable systematic support for accountability in PaaS cloud, the lists below are a discussion of applying the history of customers' critical files to assist in mitigating the risks associated with threat 1 in PaaS to benefit both sides. The application is similar to the one carried out in this thesis for IaaS.

  - Customer's critical files in PaaS provider's servers
    To provide accountability for PaaS is difficult. However, there are many customers' critical files in the provider servers. Theses files can be customer's business web application codes and database files. The database files (called datastore [150]) are provided by Google for the customers. All mentioned customers' valuable critical files are in the provider's servers. These files may be accessed without permission, by attackers caused by threat 1. This threat such as mis-using customer VMs is also relevant for the security of PaaS [23]. Thus, it is important to deal with this.
  - Threat 1 can effects both customers and providers in the PaaS environment
    Threat 1 can effect security concerns from both provider and customer perspective. For example, from the provider perspectives, [23] state that PaaS providers have traditionally suffered most from spammer, and malicious code authors, and other criminals have been able to conduct their activities with relative impunity.
    The threat can also affect the PaaS customers. For example, it is also possible that criminals may use PaaS to attack other PaaS users or the providers. For example, in order to use PaaS, customers have to upload web application codes to the provider's

severs which may potentially be shared among many customers; thus, it is possible that malicious customers may upload the codes for the purposes of compromising other customer web applications or the provider's PaaS management systems, similar to the incidents in IaaS discussed in Section 2.5.

  - History of the critical files can mitigate the risks associated with threat 1 in PaaS to the benefit of both sides
    It is difficult to deal with threat 1 for PaaS. However, PaaS can be built up by adding extra layers on top of IaaS layers as discussed in the beginning of Section 2.2. Thus, comprehension of the generic logging components and use of the history of customer's critical files for IaaS, demonstrated in this thesis, can be a guild line in mitigating the risks for PaaS. For example, after the customers upload their web application codes to provider's severs, it is possible to track the history of critical files as we did in this thesis. This may be achieved by placing loggers in PaaS management systems.

2. **Performance Analysis of Logging Systems Which are Based on the Generic Logging Components**

   This thesis presents only an example of how to analyse the security of a new logging architecture itself before deployment, in Section 3.4.2.4. However, in the complex IaaS environment, performance of the new built logging system could also be a critical factor that needs to be thoroughly analysed before deployment. For example, from the generic logging components (Figure 3.1), the performance of a logging system that deploys P2 in dom0 kernel, and of one that deploys P3 in domU kernel, should be different and needs to be considered before deployment. With knowledge of the locations of Px in the newly built logging system architecture, made clear by using the generic logging components, the system's performance analysis could be feasible. We believe that the generic logging components may be seen as a preliminary study to eventually achieve a complete analysis of the newly built logging systems, based on all possible aspects, such as security and performance.

3. **To build tools for support of file-centric logging in the cloud using introspection approaches**

   From the introspection logging approaches presented in this thesis, one of the potential related future works is to build tools to support file-centric logging (e.g., the history of critical files and process behaviour logs) for the purposes of accountability in the cloud. These tools should facilitate logging in the cloud with the introspection approaches. The tools can be an app0 or a library in the dom0 user level. The logger in this thesis can be considered as one of the primary elements to build the tools. In order to record the history of critical files and process behaviour logs of a target domU from dom0 using the introspection approaches, the tools should allow their users to specify essential configuration parameters, for example, an id or name of the target domU, and target critical files or processes in the target domU.

4. **To apply the logging approaches in this dissertation to CSA threat 2 to 7**

   One of the potential future works is to apply the logging approaches in this dissertation to CSA threat 2 to 7.

   This thesis provides logging solutions (the history of critical files and process behaviour logs) to assist in mitigating the risks associated with CSA threat 1 which is abuse and nefarious use of cloud computing. However, this thesis does not provide the solutions to alleviate CSA threats 2 to 7. Threat 2 to 7 are insecure interfaces and APIs, malicious insiders, shared technology issues, data loss or leakage, account or service hijacking, and unknown risk profile respectively, as discussed in Section 2.3.

   The logging solutions in this thesis can be applied to threat 2 to 7. We can consider that there are two common aspects for all seven CSA threats. Firstly, based on the customers' critical files in domUs (Section 5.2) discussed in this thesis, all threats can cause the same serious concern for customers. This concern is undesired accesses to the customers' critical files.

   Secondly, as discussed in Section 2.6.2, for accountability, trust, and security in cloud computing, logs are one example of the detective controls which act as psychological obstructions to go against policies or procedures in the cloud [27]. Based on the logs we have proposed in this thesis, the history of critical files and process behaviour logs in Chapter 5 and 6, these logs can represent solutions to assist in mitigating risks associated with CSA threat 2 to 7.

These logs may not be the absolute solutions. However, they can at least be a psychological barrier to prevent attackers going against policies or procedures in the cloud related to all CSA threats, as discussed above. The purpose of these logs can be seen in the same way as speed cameras serve for traffic control; the existence of speed cameras will prevent law-abiding persons from speeding, but the existence cannot halt speeding from taking place [27].

Thus, one of the potential related future works is to apply the logging approaches in this dissertation to CSA threat 2 to 7.

Bibliography

[1] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," School of Computer Science, Georgia Institute of Technology, Tech. Rep. GT-CS-11-05, 2011. [Online]. Available: http://www.bryanpayne.org/storage/GT-CS-11-05.pdf xiv, 29, 31, 32, 33, 91, 99, 114, 115, 138, 196, 197

[2] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010. xiv, 35, 90, 134, 151, 152, 155, 159, 160, 166

[3] J. C. Perez and S. Saez. (2013) Study of an operating system. [Online]. Available: http://futura.disca.upv.es/~eso/en/t3-procesos-en-linux/gen-t3-procesos-en-linux.html xiv, 151

[4] P. Mell and T. Grance, "The NIST definition of cloud computing," Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, U.S. Department of Commerce, Tech. Rep. NIST Special Publication 800-145, 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf 1

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672 1, 2, 4, 6, 9, 20, 27

[6] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, Jul.

2008. [Online]. Available: http://doi.acm.org/10.1145/1364782.1364786 2, 20

[7] G. Reese, *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud.* O'Reilly Media, 2009. [Online]. Available: http://books.google.co.uk/books?id=j8YO7gVqMqAC 2, 20

[8] A. Haeberlen, "A case for the accountable cloud," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 52–57, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773926 2, 4, 6, 8, 9, 27, 28, 29, 33, 39, 127, 145

[9] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling data in the cloud: outsourcing computation without outsourcing control," in *Proceedings of the 2009 ACM workshop on Cloud computing security.* New York, NY, USA: ACM, 2009, pp. 85–90. [Online]. Available: http://doi.acm.org/10.1145/1655008.1655020 2, 3, 4, 6, 8, 9, 28, 29, 128

[10] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proceedings of the 2009 conference on Hot topics in cloud computing.* Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855533.1855536 2, 4

[11] J. Baliga, R. Ayre, K. Hinton, and R. Tucker, "Green cloud computing: Balancing energy in processing, storage, and transport," *Proceedings of the IEEE*, vol. 99, no. 1, pp. 149–167, 2011. 2

[12] S. Mitra. (2013) Sugata Mitra: Build a school in the cloud. [Online]. Available: http://www.ted.com/talks/sugata_mitra_build_a_school_in_the_cloud.html 2

[13] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, "Provenance for the cloud," in *Proceedings of the 8th USENIX conference on File and storage technologies*, ser. FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 15–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855526 2

[14] D. Sutton, “Cloud surfing,” *iTNOW*, September 2013. [Online]. Available: http://itnow.oxfordjournals.org/content/55/3/26.full.pdf+html 2

[15] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, “Making a cloud provenance-aware,” in *First workshop on Theory and practice of provenance*. Berkeley, CA, USA: USENIX Association, 2009, pp. 12:1–12:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1525932.1525944 2

[16] clem-project.eu, “The European project Cloud Services for E-learning in Mechatronics technology (CLEM).” Online article published at The University of Manchester, 2012. [Online]. Available: http://www.clem-project.eu/ 2

[17] Amazon Web Services, Inc. (2012) Amazon elastic compute cloud (Amazon EC2). [Online]. Available: http://aws.amazon.com/ec2/ 3, 20, 122, 194

[18] R. De Paris, “FReMI-a middleware to handle molecular docking simulations of fully-flexible receptor model in hpc environment,” Master’s thesis, 2012. 3, 122, 124

[19] The University of Manchester, “Genetics ’cloud’ to create new opportunities for researchers and clinicians.” Online article published at The University of Manchester, 2011. [Online]. Available: http://www.manchester.ac.uk/aboutus/news/display/?id=7371 3

[20] K. Flanagan, S. Nakjang, J. Hallinan, C. Harwood, R. P. Hirt, M. R. Pocock, and A. Wipat, “Microbase2.0: A generic framework for computationally intensive bioinformatics workflows in the cloud,” *J. Integrative Bioinformatics*, 2012. 3

[21] K. S. Flanagan, “A grid and cloud-based framework for high throughput bioinformatics,” Ph.D. Thesis, School of Computing Science Newcastl University, 2010. 3

[22] CSA, “Security guidance for critical areas of focus in cloud computing version 3,” The Cloud Security Alliance (CSA), Tech. Rep., November 2011.

[Online]. Available: https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf 3, 20, 21, 58, 62, 221

[23] ——, "Top threats to cloud computing, version 1.0," Tech. Rep., 2010. [Online]. Available: https://cloudsecurityalliance.org/topthreatscsathreats.v1.0.pdf 3, 4, 7, 23, 24, 26, 31, 62, 125, 126, 133, 221, 222

[24] J. Zierick. (2011) Elevate cloud security with privilege delegation. [Online]. Available: http://www.ibm.com/developerworks/cloud/library/cl-datacentermigration/cl-datacentermigration-pdf.pdf 3

[25] D. Gray, R. Los, D. Shackleford, and B. Sullivan, "Top threats to cloud computing survey results update 2012," The Cloud Security Alliance (CSA), Tech. Rep., 2012. [Online]. Available: https://downloads.cloudsecurityalliance.org/initiatives/top_threats/Top_Threats_Cloud_Computing_Survey_2012.pdf 3, 7, 25, 26

[26] CSA, "The notorious nine: Cloud computing top threats in 2013," The Cloud Security Alliance (CSA), Tech. Rep., 2013. [Online]. Available: https://cloudsecurityalliance.org/download/the-notorious-nine-cloud-computing-top-threats-in-2013/ 3, 25, 26

[27] R. K. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee, "Trustcloud: A framework for accountability and trust in cloud computing," *IEEE Congress on Services*, pp. 584–588, 2011. 4, 5, 6, 8, 9, 29, 30, 32, 33, 84, 133, 181, 225

[28] J. Lyle and A. Martin, "Trusted computing and provenance: better together," in *Proceedings of the 2nd conference on Theory and practice of provenance*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855795.1855796 4, 61

[29] N. Papanikolaou, "Accountability in cloud computing: An introduction to the issues, approaches, and tools." Presentation slides published at the BIC project (a Coordination Action project funded by the European

Commission's Framework Programme 7), 2012. [Online]. Available: http://www.bic-trust.eu/files/2012/04/WG1_4_NP.pdf 4, 5

[30] R. Ko, P. Jagadpramana, and B. S. Lee, "Flogger: A file-centric logger for monitoring file access and transfers within cloud computing environments," in *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, nov. 2011, pp. 765 –771. 4, 9, 29, 30, 33, 42, 43, 44, 50, 77, 90, 93, 99, 100, 105, 113, 127, 137, 151, 166, 167, 168, 169, 171, 181, 182, 195, 196, 201

[31] O. Q. Zhang, R. K. L. Ko, M. Kirchberg, C. H. Suen, P. Jagadpramana, and B. S. Lee, "How to track your data: Rule-based data provenance tracing algorithms," in *Proceedings of 2012 International Symposium on Advances in Trusted and Secure Information Systems*, 2012. 5

[32] V. Kane, "Architecture for Managing Clouds," Distributed Management Task Force, Inc., Tech. Rep., 2010. [Online]. Available: http://www.dmtf.org/sites/default/files/standards/documents/DSP-IS0102_1.0.0.pdf 5

[33] S. Crane, "TSB Trust Domains Project," Hewlett-Packard Development Company, L.P., Tech. Rep., 2011. [Online]. Available: http://www.hpl.hp.com/research/cloud_security/Trust_Domains_Guide.pdf 5, 132

[34] Cambridge University Press . (2013) English definition of "systematic". [Online]. Available: http://dictionary.cambridge.org/dictionary/british/systematic?q=systematic 5

[35] Oxford University Press. (2013) Definition of systematic in english. [Online]. Available: http://oxforddictionaries.com/definition/english/systematic?q=systematic 5

[36] Macmillan Publishers Limited. (2013) Systematic - definition. [Online]. Available: http://www.macmillandictionary.com/dictionary/british/systematic 5

[37] R. Anderson. (1996) The trusted computing base. [Online]. Available: http://www.cl.cam.ac.uk/~rja14/policy11/node22.html 6

[38] D. G. Murray, G. Milos, and S. Hand, “Improving Xen security through disaggregation,” in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE ’08. New York, NY, USA: ACM, 2008, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346278 6, 7, 36, 62, 169

[39] B. Payne, M. de Carbone, and W. Lee, “Secure and flexible monitoring of virtual machines,” in *Twenty-Third Annual Computer Security Applications Conference, 2007. ACSAC 2007.*, 2007, pp. 385 –397. 6, 7, 8, 21, 29, 31, 32, 33, 36, 90, 91, 93, 99, 106, 107, 114, 147, 151, 167, 168, 169, 171, 194, 196, 197, 201

[40] J. M. McCune, “Reducing the trusted computing base for applications on commodity systems,” Ph.D. dissertation, School of Electrical and Computer Engineering Carnegie Mellon University Pittsburgh, PA 15213, USA, 2009. 7, 36

[41] F. Rocha and M. Correia, “Lucy in the sky without diamonds: Stealing confidential data in the cloud,” in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, ser. DSNW ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 129–134. [Online]. Available: http://dx.doi.org/10.1109/DSNW.2011.5958798 7, 36, 49, 93, 129, 131

[42] F. Rocha, T. Gross, and A. van Moorsel, “Defense-in-depth against malicious insiders in the cloud,” in *IEEE International Conference on Cloud Engineering*, 2013, pp. 88–97. 7, 36

[43] P. Macko, M. Chiarini, and M. Seltzer, “Collecting provenance via the Xen hypervisor,” in *3rd Workshop on the Theory and Practice of Provenance*, June 2011. 8, 21, 29, 30, 33, 42, 43, 44, 77, 84, 85, 110, 127, 133, 137, 139, 166, 168, 169, 170, 181, 194, 195, 196

[44] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, “Accountable virtual machines,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI’10. Berkeley,

CA, USA: USENIX Association, 2010, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924952 8, 29, 30, 32, 33, 42, 43, 44, 73, 99, 112, 168, 195, 196, 201

[45] R. Ko, M. Kirchberg, and B. S. Lee, "From system-centric to data-centric logging-Accountability, trust & security in cloud computing," in *Defense Science Research Conference and Expo (DSR), 2011*, 2011, pp. 1 –4. 9, 28, 181

[46] J. Spring, "Monitoring cloud computing by layer, part 1," *Security & Privacy, IEEE*, vol. 9, no. 2, pp. 66 –68, 2011. 9, 21, 49, 58, 59, 89, 98, 104, 131, 221

[47] ——, "Monitoring cloud computing by layer, part 2," *Security Privacy, IEEE*, vol. 9, no. 3, pp. 52–55, 2011. 9

[48] G. Aceto, A. Botta, W. de Donato, and A. Pescape, "Cloud monitoring: Definitions, issues and future directions," 2012. 9

[49] P. Kuchana, *Software Architecture Design Patterns in Java.* Boston, MA, USA: Auerbach Publications, 2004. 10, 81, 82

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994. 10, 81, 84

[51] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462 11

[52] F. Rocha, S. Abreu, and M. Correia, "The final frontier: Confidentiality and privacy in the cloud," *Computer*, vol. 44, pp. 44–50, 2011. 15, 23, 55, 60, 98, 191

[53] W. Wongthai, F. Rocha, and A. van Moorsel. A generic logging architecture template to mitigate the risks associated with spam activities in infrastruc-

ture as a service cloud. In Poster Session at The Newcastle Connection 2012, August, Newcastle, UK. 17

[54] ——, "A generic logging template for infrastructure as a service cloud," in *Proceedings of the 2013 27th International Conference on Advanced Information Networking and Applications Workshops*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1153–1160. [Online]. Available: http://dx.doi.org/10.1109/WAINA.2013.108 17

[55] W. Wongthai, F. Rocha, and A. Van Moorsel, "Logging solutions to mitigate risks associated with threats in infrastructure as a service cloud," in *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*, Dec 2013, pp. 163–170. 17

[56] Google. Google app engine. [Online]. Available: https://developers.google.com/appengine/ 20

[57] W. Chun. (2012) What is cloud computing? [Online]. Available: https://developers.google.com/appengine/training/intro/whatiscc 21

[58] W. Dawoud, I. Takouna, and C. Meinel, "Infrastructure as a service security: Challenges and solutions," in *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, march 2010, pp. 1–8. 21, 30, 130

[59] H. Fang, Y. Zhao, H. Zang, H. Huang, Y. Song, Y. Sun, and Z. Liu, "VM-guard: An integrity monitoring system for management virtual machines," in *International Conference on Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th*, 2010, pp. 67–74. 21, 130, 131, 194

[60] X. Jing and Z. Jian-jun, "A brief survey on the security model of cloud computing," in *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on*, 2010, pp. 475–478. 21

[61] M. Hogan, F. Liu, A. Sokol, and J. Tong, "NIST cloud computing standards roadmap," Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, U.S.

Department of Commerce, Tech. Rep. NIST Special Publication 800-145, 2011. [Online]. Available: http://www.nist.gov/itl/cloud/upload/NIST_SP-500-291_Jul5A.pdf 21

[62] R. Selie. (2012) What is the impact of cloud to the architecture of a solution? Xebia. [Online]. Available: http://blog.xebia.com/2012/08/21/what-is-the-impact-of-cloud-to-the-architecture-of-a-solution/ 21

[63] CSA. About the cloud security alliance. Cloud Security Alliance. [Online]. Available: https://cloudsecurityalliance.org/about/ 23

[64] W. Ashford. (2011) Rsa 2011: Emc announces rsa cloud trust authority for mission-critical applications. [Online]. Available: http://www.computerweekly.com/news/1280095155/RSA-2011-EMC-announces-RSA-Cloud-Trust-Authority-for-mission-critical-applications 24

[65] W. Claycomb and A. Nicoll, "Insider threats to cloud computing: Directions for new research challenges," in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, 2012, pp. 387–394. 24

[66] Malware Domain List. (2009) Malware domain list. [Online]. Available: http://www.malwaredomainlist.com/ 25

[67] D. Danchev. (2009) Zeus crimeware using Amazon's EC2 as command and control server. CBS Interactive. [Online]. Available: http://www.zdnet.com/blog/security/zeus-crimeware-using-amazons-ec2-as-command-and-control-server/5110 25

[68] Cisco Systems. (2013) ScanSafe is now part of Cisco. Cisco Systems, Inc. [Online]. Available: http://www.cisco.com/web/about/ac49/ac0/ac1/ac259/scansafe.html 25

[69] B. Krebs. (2008) Amazon: Hey spammers, get off my cloud! The Washington Post. [Online]. Available: http://voices.washingtonpost.com/securityfix/2008/07/amazon_hey_spammers_get_off_my.html 25

[70] L. Leong. (2011) There's no such thing as a "safe" public cloud IaaS. Insecure.Org. [Online]. Available: http://blogs.gartner.com/lydia_leong/2011/11/11/theres-no-such-thing-as-a-safe-public-cloud-iaas/ 25

[71] D. Kerr. (2012) Pirate bay ditches servers and switches to the cloud. CBS Interactive Inc. [Online]. Available: http://news.cnet.com/8301-1023_3-57534707-93/pirate-bay-ditches-servers-and-switches-to-the-cloud/ 25

[72] K. Khan and Q. Malluhi, "Establishing trust in cloud computing," *IT Professional*, vol. 12, no. 5, pp. 20–27, 2010. 26

[73] Macmillan Publishers Limited. (2013) accountability - definition. [Online]. Available: http://www.macmillandictionary.com/dictionary/british/accountability 27

[74] C. Modi, D. Patel, B. Borisaniya, A. Patel, and M. Rajarajan, "A survey on security issues and solutions at different layers of cloud computing," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 561–592, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11227-012-0831-5 27, 131

[75] S. Nakahara and H. Ishimoto, "A study on the requirements of accountable cloud services and log management," in *Information and Telecommunication Technologies (APSITT), 2010 8th Asia-Pacific Symposium on*, june 2010, pp. 1 –6. 28

[76] S. Sundareswaran, A. C. Squicciarini, and D. Lin, "Ensuring distributed accountability for data sharing in the cloud," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, pp. 555–567, 2012. 29, 31, 42, 73, 91, 105, 111, 124, 127, 137, 181, 182, 195, 196

[77] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy 2008. SP 2008.*, 2008, pp. 233–247. 31, 110, 197, 198

[78] A. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "Cloudsec: A security monitoring appliance for virtual machines in the iaas cloud model," in

2011 5th International Conference on Network and System Security (NSS), 2011, pp. 113–120. 32, 33, 48, 90, 92, 93, 105, 128, 151, 167

[79] B. Payne. (2013) About the VMI tools project. Google Project Hosting. [Online]. Available: http://code.google.com/p/vmitools/ 32, 33, 42, 52, 55, 90, 98, 105, 149, 150, 178, 201, 205

[80] T. Burghardt, K. Böhm, A. Guttmann, and C. Clifton, "Anonymous search histories featuring personalized advertisement-balancing privacy with economic interests," *Transactions on Data Privacy*, vol. 4, no. 1, pp. 31–50, Apr. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2019312.2019315 34

[81] B. Parno, J. M. McCune, and A. Perrig, *Bootstrapping trust in modern computers*. Springer, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-1460-5 36

[82] M. Zhou, R. Zhang, W. Xie, W. Qian, and A. Zhou, "Security and privacy in cloud computing: A survey," in *Semantics Knowledge and Grid (SKG), 2010 Sixth International Conference on*, 2010, pp. 105–112. 39

[83] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *In Proceedings of the 18th USENIX Security Symposium*, 2009. 39, 59, 61, 91, 98, 105

[84] The Linux Information Project (LINFO), "Kernel space definition," http://www.linfo.org/kernel_space.html. 45

[85] L. Soares and M. Stumm, "FlexSC: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924946 46

[86] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004. 47

[87] University of Miami, "Confidentiality, integrity and availability (CIA)," http://it.med.miami.edu/x904.xml, 2008. 48

[88] D. S. Wall, "Organizational security and the insider threat: Malicious, negligent and well-meaning insiders," 2011. [Online]. Available: https://deloitte.symantec.com/system/files/Organizational%20Security%20and%20the%20InsiderThreat%20Malicious%20Negligent%20and%20Well-Meaning%20Insiders_WP_FINAL_012811.pdf 49, 89, 98, 104

[89] A. Baldwin and S. Shiu, "Managing digital risk: Trends, issues, and implications for business," Lloyd's, Tech. Rep. Lloyds 360 Risk Insight. 49, 89, 98, 104

[90] S. Stolfo, M. Salem, and A. Keromytis, "Fog computing: Mitigating insider data theft attacks in the cloud," in *2012 IEEE Symposium on Security and Privacy Workshop (SPW)*, 2012, pp. 125–128. 49

[91] S. Ludwig. Cloud 101: What the heck do iaas, paas and saas companies do? [Online]. Available: http://venturebeat.com/2011/11/14/cloud-iaas-paas-saas/ 58, 221

[92] S. E. Parkin and G. Morgan, "Toward reusable sla monitoring capabilities," *Softw. Pract. Exper.*, vol. 42, no. 3, pp. 261–280, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1060 61, 85

[93] F. Beck and O. Festor, "Syscall Interception in Xen Hypervisor," Inria, Technical Report, 2009. [Online]. Available: http://hal.inria.fr/inria-00431031 73, 85, 99, 170

[94] J. Heer and M. Agrawala, "Software design patterns for information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 853–860, 2006. 81, 83

[95] F. Gadaleta, N. Nikiforakis, J. T. Mühlberg, and W. Joosen, "Hyperforce: Hypervisor-enforced execution of security-critical code," *CoRR*, vol. abs/1405.5648, 2014. 85, 98

[96] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger, "Cloud verifier: Verifiable auditing service for iaas clouds," in *203 IEEE Ninth World Congress on Services (SERVICES)*, 2013, pp. 239–246. 90, 93, 151, 167, 171

[97] C. Maiero and M. Miculan, "Unobservable intrusion detection based on call traces in paravirtualized systems," in *Proc. SECRYPT*, 2011. [Online]. Available: http://sole.dimi.uniud.it/~marino.miculan/Papers/SECRYPT11.pdf 90, 93, 151, 167

[98] R. Anderson, F. Stajano, and J.-H. Lee, "Security policies," ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2002, vol. 55, pp. 185 – 235. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245801800309 92

[99] M. A. Bishop, *Computer Security. Art and Science.* Addison-Wesley Professional, 2002. 92

[100] S. B. Lipner, "Non-discretionary controls for commercial applications," in *IEEE Symposium on Security and Privacy'82*, 1982, pp. 2–10. 92

[101] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept 1975. 106

[102] Red Hat, Inc. (2013) Red Hat Enterprise Linux 6 Deployment Guide Deployment, Configuration and Administration of Red Hat Enterprise Linux 6 Edition 5. [Online]. Available: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/ch-Managing_Users_and_Groups.html 121

[103] The Linux Documentation Project (LDP). (2013) Linux system administrators guide: Chapter 11. managing user accounts. The Linux Documentation Project (LDP). [Online]. Available: http://www.tldp.org/LDP/sag/html/account.html 121

[104] Fedora Documentation. (2013) Set the root password. Fedora Documentation. [Online]. Available: http://docs.fedoraproject.org/en-US/Fedora/13/html/Installation_Guide/sn-account_configuration.html 121

[105] Red Hat, Inc. (2013) Administration guide draft/useraccounts. Red Hat, Inc. [Online]. Available: https://fedoraproject.org/wiki/Administration_Guide_Draft/UserAccounts?rd=Docs/Drafts/AGBeta/UserAccounts 121, 122, 123, 185

[106] Red Hat, Inc. and others., "Fedora," http://fedoraproject.org/, 2013. 121, 122

[107] A. Mihoob, C. Molina-Jimenez, and S. Shrivastava, "A case for consumer," *2012 IEEE Fifth International Conference on Cloud Computing*, vol. 0, pp. 506–512, 2010. 122

[108] Datapipe Cloud Reports. (2013) How to separate key-pair authentication for individual users. Newvem Insight Ltd. [Online]. Available: http://www.newvem.com/how-to-separate-key-pair-authentication-for-separate-aws-iam-users/ 122, 123, 124, 190

[109] Amazon Web Services, Inc. (2013) Amazon EC2 Running Red Hat Enterprise Linux. Amazon Web Services, Inc. [Online]. Available: http://aws.amazon.com/rhel/ 122

[110] G. Garron. (2013) EC2 Red Hat : SSH Add User with Password. YouTube, LLC. [Online]. Available: http://www.youtube.com/watch?v=hDVCP1VFSTY 123

[111] Amazon Web Services, Inc. (2013) Amazon ec2 key pairs. Amazon Web Services, Inc. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html 124

[112] European Commission, "Towards a general policy on the fight against cyber crime," European Commission, Tech. Rep. COM(2007) 267 final. 125

[113] R. Anderson, C. Barton, R. Böhme, R. Clayton, M. van Eeten, M. Levi, T. Moore, and S. Savage, "Measuring the cost of cybercrime," in *WEIS*, 2012. [Online]. Available: http://lyle.smu.edu/~tylerm/weis12.pdf 126

[114] Detica and Office of Cyber Security and Information Assurance, "The cost of cyber crime," https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/60943/the-cost-of-cyber-crime-full-report.pdf, February 2011. 126

[115] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A multifaceted approach to understanding the botnet phenomenon," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, ser. IMC '06. New York, NY, USA: ACM, 2006, pp. 41–52. [Online]. Available: http://doi.acm.org/10.1145/1177080.1177086 126

[116] Conjecture Corporation. (2013) What is rsa encryption? [Online]. Available: http://www.wisegeek.com/what-is-rsa-encryption.htm 127

[117] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler, "Detecting co-residency with active traffic analysis techniques," in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, ser. CCSW '12. New York, NY, USA: ACM, 2012, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2381913.2381915 128

[118] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing," 2005, zyb@is.iscas.ac.cn 13083 received 27 Oct 2005. [Online]. Available: http://eprint.iacr.org/2005/388 128

[119] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 305–316. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382230 128

[120] Free Software Foundation, Inc. (2013) Libgcrypt. http://www.gnu.org/software/libgcrypt/. 128

[121] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds,” in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS ’09. New York, NY, USA: ACM, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687 129

[122] CSA. (2010) Cloud Security Alliance and HP Identify Top Cloud Security Threats in New Research Report. [Online]. Available: https://cloudsecurityalliance.org/pr20100301a.html 129

[123] K. J. Higgins, “Hacking tool lets a vm break out and attack its host,” http://www.darkreading.com/applications/hacking-tool-lets-a-vm-break-out-and-att/217701908. 130

[124] D. Jaeger, K.-F. Krentz, and M. Richly, “Organizational security and the insider threat: Malicious, negligent and well-meaning insiders,” http://www.hpi.uni-potsdam.de/fileadmin/_migrated/content_uploads/The_Guests_still_Strike_Back.pdf. 130

[125] D. Chisnall, *The definitive guide to the Xen hypervisor*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. 130

[126] W. Kandek and D. Reading. (2010) Virtualization vulnerabilities up and coming. UBM Tech. [Online]. Available: http://www.darkreading.com/blog/227700989/virtualization-vulnerabilities-up-and-coming.html 130, 131

[127] P. S. Wooley. (2011) Identifying cloud computing security risks. University of Oregon. [Online]. Available: https://scholarsbank.uoregon.edu/xmlui/bitstream/handle/1794/11393/Wooley-2011.pdf 130

[128] The Linux Information Project. (2006) Malware definition. Bellevue Linux. [Online]. Available: http://www.linfo.org/malware.html 130

[129] K. Onoue, Y. Oyama, and A. Yonezawa, “Control of system calls from outside of virtual machines,” in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC ’08. New York,

NY, USA: ACM, 2008, pp. 2116–1221. [Online]. Available: http://doi.acm.org/10.1145/1363686.1364196 131

[130] K. Borders, E. V. Weele, B. Lau, and A. Prakash, "Protecting confidential data on personal computers with storage capsules," in *Proceedings of the 18th conference on USENIX security symposium*, 2009. 131

[131] A. Bisong and S. S. M. Rahman, "An overview of the security concerns in enterprise cloud computing," *International Journal of Network Security & Its Applications (IJNSA)*, vol. 3, no. 1, Jan. 2011. [Online]. Available: http://airccse.org/journal/nsa/0111jnsa03.pdf 132, 145, 192

[132] A. Saha. (2006) Learning about linux processes. The Linux Gazette. [Online]. Available: http://linuxgazette.net/133/saha.html 138, 152

[133] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd ed. USA: Addison-Wesley Publishing Company, 2010. 138

[134] T. Jones. (2008) Anatomy of linux process management. [Online]. Available: http://www.ibm.com/developerworks/library/l-linux-process-management/ 139

[135] D. A. Rusling, *The Linux kerel*. David A Rusling 3 Foxglove Close, Wokingham, Berkshire RG41 3NF, United Kingdom: David A Rusling, 1999. [Online]. Available: http://www.tldp.org/LDP/tlk/ 149, 155, 159, 160

[136] A. Kissner. (2006) Understanding a process information pseudo-file system. Linux Forums. [Online]. Available: http://www.linuxforums.org/articles/understanding-proc_82.html 152

[137] The LXR team. (2013) The lxr project web-site. The LXR team. [Online]. Available: http://lxr.sourceforge.net/en/index.shtml 152

[138] I. Wienand. (2013) File descriptors. [Online]. Available: http://www.bottomupcs.com/file_descriptors.html 155

[139] Sascha Nitsch Management Consulting UG. (2013) www.linuxhowtos.org howtos, tips&tricks and tutorials for linux. [Online]. Available: http://www.linuxhowtos.org/data/6/fd.txt 155

[140] The Linux Documentation Project (LDP). (2013) The linux kernel module programming guide. The Linux Documentation Project (LDP). [Online]. Available: http://www.tldp.org/LDP/lkmpg/2.4/html/c577.htm 159

[141] S. Pillai. (2012) inode and its structure in linux. [Online]. Available: http://slashroot.in/inode-and-its-structure-linux 160

[142] linux-faqs.info. (2013) Difference between mtime, ctime and atime. [Online]. Available: http://www.linux-faqs.info/general/difference-between-mtime-ctime-and-atime 160

[143] linux.die.net. (2010) rm(1)-linux man page. [Online]. Available: http://linux.die.net/man/1/rm 163

[144] blogspot.co.uk. (2008) linux device driver (part 1) process, system call, kernel and hardware. [Online]. Available: http://hsuedw.blogspot.co.uk/2008/08/linux-device-driver-part-1-process.html 166

[145] H. Arora. (2012) Understanding linux open() system call. The International Business Machines Corporation (IBM). [Online]. Available: https://www.ibm.com/developerworks/mydeveloperworks/blogs/58e72888-6340-46ac-b488-d31aa4058e9c/entry/understanding_linux_open_system_call?lang=en 170

[146] Rackspace, US Inc. (2013) Build what you want. Rackspace, US Inc. [Online]. Available: http://www.rackspace.com/cloud/ 194

[147] B. Payne. Xenaccess, a virtual machine introspection library for Xen. [Online]. Available: http://code.google.com/p/xenaccess/ 201

[148] Linux Documentation. (2013) usleep(3) - linux man page. Die.net. [Online]. Available: http://linux.die.net/man/3/usleep 204

[149] D. Sanderson, *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*, 1st ed. O'Reilly Media, Inc., 2009. 221

[150] Google Inc. (2013) Python datastore API. Google Inc. [Online]. Available: https://developers.google.com/appengine/docs/python/datastore/ 222